Component technology in an embedded system

David Polberger

Thesis for a diploma in computer science, 30 ECTS credits, Department of Computer Science, Faculty of Science, Lund University

Examensarbete för 30 hp, Institutionen för datavetenskap, Naturvetenskapliga fakulteten, Lunds universitet

Component technology in an embedded system

Abstract

Software components have been touted as the solution to the modularity problems faced in the software industry, one that also gives rise to a sophisticated market of software parts. With components, proponents claim, software is effortlessly built by combining components readily available for procurement. This master's thesis examines components from a technical perspective and brings into focus the industry standards that enable interoperability between components. Particular attention is given to freestanding interfaces, and an object model supporting this concept is developed in the C programming language.

This thesis also includes a discussion of the component technology developed at ST-Ericsson and Sony Ericsson for use in their embedded systems. An execution tracing facility for this environment, enabled using a declarative attribute, is presented, along with a discussion of services customized through declarative means in the enterprise.

Komponentteknik i ett inbyggt system

Sammanfattning

Mjukvarukomponenter har förts fram som lösningen på problemet med att sätta samman mjukvara från fristående delar. Teknikens förespråkare hävdar att program enkelt kan byggas genom att kombinera komponenter som köps in på en marknad. Detta examensarbete studerar mjukvarukomponenter från ett tekniskt perspektiv och diskuterar de industristandarder som gör det möjligt för komponenter att samverka. Särskild vikt läggs vid fristående gränssnitt, och en objektmodell som stöder sådana utvecklas i programmeringsspråket C.

Detta arbete studerar även den komponentteknik som utvecklats vid ST-Ericsson och Sony Ericsson för användning i deras inbyggda system. En tjänst som aktiveras genom ett deklarativt attribut och som möjliggör spårning av körtidsaktivitet har utvecklats som en del av detta examensarbete, och dess implementation presenteras. En allmän diskussion av implementationen av tjänster som konfigureras genom deklarativa attribut, företrädesvis i industriell klient-server-miljö, ingår också.

Typeset using LaTeX 2_ε with the Memoir document class. Figures drawn in Inkscape.

Contents

Figures

Listings

Preface

Software components are units of composition, from which component-based software is built. **Software components are little more than classes.** Software components are similar to components in other engineering disciplines, and may thus be sold on a market, competing on price and functionality. **Software components are "better libraries" that declaratively state their version information and their dependencies, thus avoiding "DLL hell."** Software components are written by skilled programmers and assembled by personnel with a different set of skills. **Software components are containers of classes, from which objects are instantiated.** Software components succeed where objects failed. **Software components encourage black box reuse, as opposed to white box reuse.** Software components are manipulated in visual designers and customized through their properties. **Software components run within the confines of application servers in the enterprise.** Software components revolutionize the construction of software. **Software components play by well-defined rules that govern their memory usage, names, runtime type information and object invocations.** Software components encapsulate business logic in the enterprise. **Software components are usable in binary form, and do not require access to their source code.** Software components and component-oriented programming herald a new era in software reuse. **Software components request services from their containers in a declarative manner.** Software components may be composed in the same way that LEGO bricks can be used to build new structures. **Software components may be substituted for one another due their use of interfaces and dynamic dispatch.** Software components require new business processes. **Software components' role as reusable entities is due to strict adherence to standards.** Software components are rigorously documented entities that are certified by trusted, third-party organizations. **Software components are stateless entities that may be independently deployed without modification.** Software components can be used as a middle ground alternative to writing software from scratch and using fully ready-made solutions. **Software components are independent, encapsulated software entities.** Software components can be anything, from database servers to source code snippets. **Software components enable the seamless, effortless use of services running on other computers.**

Introduction

Much has been written about software components over the years. The sampling on the preceding page reflects some of the many disparate and often contradictory lines of thought that I have encountered while working on this thesis. While there is considerable discrepancy of opinion over some of the finer points of the definition of software components, there is universal agreement that the goal of the discipline is to make it possible to create software, partly or fully, from prefabricated parts. Software components, in other words, facilitate software reuse.

This discipline goes by many names. Some refer to designing software with components as "Component-Based Design" (CBD) and to the branch of software engineering as "Component-Based Software Engineering" (CBSE). Others prefer "software componentry," or "component technology." I have elected to use the latter name in this thesis.

From a technical perspective, component technology lays down standards that enable software parts to be usable from many different environments. While software reuse in and of itself cannot be credited to component technology, the discipline does bring increased rigor. Its standards enable objects—and not just procedures—created in different environments to communicate, while insisting that interfaces are kept separate from their implementations. Some of the technology in this space bring additional features, which enable things like distributed computing and out-of-band services through declarative attributes.

The other major perspective that is often applied to component technology is the business perspective. Because component technology makes reuse possible on a grander scale than before, there is a larger market for software components than for, say, language- and vendor-specific class libraries. As a result, some envision the formation of large software component markets, offering a wide range of compatible components competing on price and functionality, ultimately leading to the transformation of the entire industry. This thesis pays very little attention to the business perspective, and almost exclusively focuses on the enabling technology.

Background

In June 2005, I started an internship at Sony Ericsson Mobile Communications AB in Lund, Sweden, with the intent of writing my master's thesis in computer science at the company. Sony Ericsson, a maker of cellular phones, arranged for me to work on their in-house-developed component technology ECMX as part of the company's software architecture group. The goal of my work was to enable developers to debug software written using ECMX, with a particular focus on invocations spanning multiple processes.

I was hired shortly after completing the practical part of my thesis work, and spent more than two years as a full-time employee with the company. During my tenure, I had the privilege of designing a new user-facing application from the ground up, which presented large quantities of hierarchically organized information related to various disparate domains. The application, which was designed to be agnostic to the nature of the content it displayed, relied on a number of ECMX objects, written by other groups, which implemented interfaces mandated by the application. Some objects were implemented in C, and others in Java, unbeknownst to the application. This application made heavy use of ECMX and its support for location-transparent invocations, giving me ample time to study its internals in the course of my professional duties, a luxury afforded few students. I left the company in March 2008 to start a software business, and, having gotten the company off the ground, wrote this report in the first half of 2009.

ECMX piqued my interest in component technology, which led me to this work. While one aim of this thesis is to present the work I did in Sony Ericsson's architecture group, another goal is to place ECMX in a historical and technical perspective, as well as to give an overview of this field and the industrial technologies associated with it. In the course of this thesis, I hope to strip component technology of its veneer of complexity, by demonstrating the straight-forward technology that powers object invocations and by discussing the implementation of component models.

Biases

I bring my biases to this work, although I have tried to be mindful of them. While I have tried to give the technologies I cover fair treatment, there is no escaping the fact that I had significant exposure to a few of them before I wrote this thesis, and fairly little experience with the others. I would be surprised if this has not affected the end result. In particular, I have spent many years with Embarcadero's Delphi product, and have written many components for it, including a rich text label that I released as open source in 2002.[1,2] I also had some experience with procuring and using third-party COM components as part of an application I worked on in the late 1990s. As a Java developer, I also had some experience with OSGi prior to writing this report.

Organization

This thesis is meant to be read sequentially—later chapters build on the material in chapters preceding them.

Chapter 1: Confronting the software crisis discusses the software crisis debate of the late 1960s, and the role of software components as a possible remedy. Components, as they are thought of today, are discussed, and a definition is settled on. Object-oriented programming in the component context is also covered, as is the role of components in enterprise computing.

Chapter 2: Realizing software components examines the technical means used to realize components more concretely. In so doing, it categorizes software components into three distinct categories, and introduces most of the concepts associated with component technology.

Chapter 3: Demystifying dynamic dispatch explores the technology behind *dynamic dispatch*. This concept is of paramount importance to component technology, as it makes it possible to program to a specification, and defer binding to an implementation until runtime. A simple object model supporting dynamic dispatch is built in the C programming language over the course of this chapter.

[1] Most of the components I have written for Delphi have been statically linked with the software using them, and as a result they cannot be considered components per the definitions of Chapter 1. Had I opted to deploy them as Delphi runtime packages instead (described on page 81), they would largely have been in compliance with these definitions.

[2] The rich text label I wrote was contributed to the JEDI Visual Component Library as `TJvLinkLabel`. To this day, Project JEDI actively maintains it, and it has spawned a fork in the `DIHtmlLabel`, which replaces the parser with one that is Unicode-aware. (A "fork" is a separate branch of a software project, one that often veers off in a different direction.)

Chapter 4: Refining the object model builds on the material in Chapter 3, and introduces more sophisticated memory management, better support for interfaces, as well as provisions for scripting languages. Constructs left out from the object model are discussed, as are the additions one could make to this object model to create a full-fledged component model consistent with the definitions of Chapter 1.

Chapter 5: Ways of the industry covers the major component-related technologies that have appeared in industry, technologies that have shaped our notion of what a software component is. Topics include Microsoft's Visual Basic, COM and .NET, Embarcadero's Delphi, OMG's CORBA, as well as OSGi for Java.

Chapter 6: The (Sony) Ericsson way introduces the component technology developed at ST-Ericsson and Sony Ericsson for use in their embedded systems. The technology created at Sony Ericsson for inter-process communication and interoperability between Java and native code is discussed at some length.

Chapter 7: Implementing interception expands on the role of declarative attributes in component technology, which are used to configure services. Implementations in industry are discussed, followed by a presentation of an execution tracing facility I implemented at Sony Ericsson, configured declaratively. Provisions for generating UML interaction sequence diagrams from traces are examined, before a discussion of opportunities for future work.

License

Contact information

You are welcome to get in touch with me, at `david@polberger.se`. This thesis is available on the Web, at `http://www.polberger.se/components/`. This site also contains the source code for Chapter 3 and Chapter 4, as well as a blog, a presentation, a browsable Web version of this thesis and potentially errata.

Acknowledgments

I would like express my gratitude to Marcus Offesson for reading a draft copy of this thesis, and for giving me valuable feedback. I would also like to thank my advisors, Ferenc Belik and Göran Fries of the Department of Computer Science and Henrik Sundström of Sony Ericsson, for their patience. It has been a pleasure working with Ola Hedbäck, the librarian at the Department of Computer Science. I would also like to acknowledge the members of the Visual Basic, COM, Delphi, OSGi and .NET communities that have reviewed Chapter 5. Last but not least, I would like to thank Marianne for her support during all these years I have spent (not) working on this thesis.

David Polberger Lund, August 2009

CHAPTER 1

Confronting the software crisis

The notion of a *software crisis*, or a *software gap*, emerged at the end of the 1960s. It was believed that the accomplishments of software fell far short of its ambitions, in terms of user expectations, performance and cost (David and Fraser, quoted in Naur and Randell 1969:120). The crisis stemmed from the difficulties encountered when building large, complex systems. Hardware was evolving at an unprecedented pace at the time, a pace software was not able to match. Edsger W. Dijkstra brought up the subject when giving a speech accepting the ACM Turing Award in 1972:

> [The primary cause of the software crisis is] that the machines have become several orders of magnitude more powerful! To put it quite bluntly: as long as there were no machines, programming was no problem at all; when we had a few weak computers, programming became a mild problem, and now [that] we have gigantic computers, programming has become an equally gigantic problem. In this sense the electronics industry has not solved a single problem, it has only created them, it has created the problem of using its products. To put it another way: as the power of available machines grew by a factor of more than a thousand, society's ambition to apply these machines grew in proportion [...]. The increased power of the hardware, together with the perhaps even more dramatic increase in its reliability, made solutions feasible that the programmer had not dared to dream about a few years before. And now, a few years later, he *had* to dream about them and, even worse, he had to transform such dreams into reality! Is it a wonder that we found ourselves in a software crisis?

A NATO-sponsored software engineering conference was held in Germany in 1968, and the purported crisis figured prominently in the discussions. The very term "software engineering" was provocatively coined for this conference—it was argued that software development was not yet a mature branch of engineering, and that the field had to evolve to earn the engineering label (Seidman 2008; Naur and Randell 1969:13).

Software reuse was not in wide use at the time. A section in the venerable magazine Communications of the ACM was dedicated to disseminating algorithms in the 1960s, but only in source code form (written in the programming language ALGOL 60), and the algorithms were meant to be adapted manually to the target language and machine (Perlis 1966). At the time, developers constantly reinvented the wheel when building systems.

In an effort to counter the crisis, Douglas McIlroy (1969:138) introduced the notion of *software components* in an invited address at the NATO conference. Instead of developers reinventing basic functionality with each new software project, reusable software components would be used instead, in much the same way the hardware industry was using pre-fabricated components:

> We undoubtedly produce software by backward techniques. We undoubtedly get the short end of the stick in confrontations with hardware people because they are the industrialists and we are the crofters. Software production today appears in the scale of industrialization somewhere below the more backward construction industries. I think its proper place is considerably higher, and would like to investigate the prospects for mass-production techniques in software.

McIlroy lamented that software developers started software projects thinking about what to *build* rather than what to *use*. To provide these ready-made components, he advocated that a software components industry be founded, providing best-of-breed software parts:

> [The] purchaser of a component [...] will choose one tailored to his exact needs. He will consult a catalogue offering routines in varying degrees of precision, robustness, time-space performance, and generality. He will be confident that each routine in the family is of high quality—reliable and efficient.

McIlroy's understanding of software components was quite different from the contemporary understanding. Components in McIlroy's vein were akin to the procedural libraries of today, and could be distributed in source code form only, contrary to the modern expectation that components may be distributed in binary form.

Another leading light in this field is Brad Cox, who introduced the *Software IC* (integrated circuit) concept in middle of the 1980s (Persson 2002:35). In contrast to McIlroy's understanding of components, a Software IC was to be available in binary form. Cox's ideas received much attention, but the Software IC concept was marred by the requirement that all components be written in the Objective-C programming language, also designed by Cox.

1.1 Putting the software component idea to work

In his address, McIlroy listed a number of software categories that would be suitable for components: mathematical functions, input-output conversion, geometry, text processing and storage management (McIlroy 1969:144). As Persson (2002:31) points out, the C standard library, which McIlroy was instrumental in creating during his time at Bell Laboratories, had routines for all of the original categories but the geometry one.

McIlroy also invented the *pipeline* mechanism (Ritchie 1980). A pipeline can be seen as a number of software components working in tandem, each serving a very specific purpose, and each unaware of the inner workings of the other components in the pipeline. A user, or a script, strings together such components, enabling complete programs to be built. The pipeline in the Unix operating system was McIlroy's first application of the concept.

More formally, a pipeline is an ordered collection of software elements that consume data from the element directly preceding them in the pipeline and produce data based on the consumed data (the first element consumes no data). A pipeline can have an arbitrary number of elements, with data flowing from the first element to the last. In Unix, each software element is a stand-alone command-line program.

For example, consider the following program that calculates the number of files and directories in the current directory. It consists of the following input to a Unix shell:[1]

```
ls | wc -l
```

The `ls` program lists all available files in the current directory. Instead of displaying this list on-screen, the pipe symbol ("`|`") causes it to be redirected to the `wc` program, in effect gluing the two programs together. The `wc` program counts the number of lines in the input data when given the `-l` argument. As `wc` is the last program in the pipeline, its output by default appears on-screen. In effect, a program has been constructed from two components that displays the number of files in a directory with very little effort.

Pipelines can be quite complex; consider the following program that identifies the largest file in the current directory:

```
ls -s | sort -n | tail -1 | awk '{print $2}'
```

The familiar `ls` program is given an `-s` argument, prompting it to produce a list of all files in the current directory, with file sizes in the first column and file names in the second column. The `sort` program is given an `-n` argument, instructing it to sort its input data numerically, resulting in a list with the smallest file at the top and the largest file at the bottom. The `tail` program extracts the last *n* lines (where *n* is one in this case, a number conveyed by the `-1` argument). What remains is a string with the size of the largest file in the first column, and its name in the second column. Finally, the `awk` program[2] extracts the contents of the second column and displays it on-screen. In effect, a program has been constructed that identifies the name of the largest file in the current directory, again with the help of reusable components in the guise of standard command-line programs.

The holy grail of component-oriented programming is to enable the same ease-of-use when creating much larger programs consisting of a wide variety of components.

1.2 Contemporary components

The software components of today have the same overarching goal as the components advocated in years past: making it possible to encapsulate discrete functionality into reusable entities. The concept has evolved considerably since the late 1960s, and today, some or all of the following should be true for a contemporary component:

- **A component stands alone.** As such, a component should not have been produced for any one project, and should prove itself useful in a multitude of projects (Vigder 2001).

[1] While these pipeline examples are meant to illustrate a concept in use since the 1970s, they have been tested on a system running a modern Linux distribution and may thus not work on a vintage Unix system.

[2] AWK is programming language designed to process strings. Its programs are interpreted by the `awk` program.

From an economic point of view, Szyperski et al. (2002) estimate that a component needs to be used in three distinct project before it breaks even. As a stand-alone entity, a component may be deployed to an end-user's system separately from the programs that make use of it. A component should not be tied to a specific programming environment or language.

- **A component is encapsulated.** The inner workings of a component are hidden, meaning that a component may only be accessed in carefully controlled ways (Sommerville 2007). This enables information hiding on a high level, in the tradition of Parnas (1972). A component is thus used without relying on its source code, which is known as *black box reuse* (as opposed to *white box reuse*). The use of black box reuse is said to stimulate the move from monolithic systems to modular ones (Szyperski et al. 2002:xxii).

- **A component is customizable.** Components should make it possible to customize their behavior, enabling them to be used in a variety of contexts.

- **A component is documented.** Aside from including developer documentation, Szyperski et al. (2002:470) suggest that the documentation should be machine-readable, making it possible to store it as part of component catalogs, which can be searched through in pursuit of suitable components. The documentation should include the component's requirements on the underlying platform and on other components—in other words, its dependencies.

- **A component is certified.** Some commentators, such as Booch et al. (2001) and Bachmann et al. (2000), have suggested that in order for components to be truly trustworthy, they need to be certified to conform to rigorous standards set by neutral, third-party organizations.

Software components are meant to enable reuse, making it possible for developers to glue interoperable software parts together (this is often referred to as *composition*). As a result, it is thought that software components create new roles in the software industry, that of component writers and component assemblers (Vitharana 2003; Szyperski et al. 2002:495). The former category is made up of skilled programmers, working with existing tools and languages, while the latter category is comprised of people with a different set of skills. Component assemblers need not use the same tools and languages used by component writers, and may instead use scripting languages in conjunction with visual designers to put together complete solutions.[3]

Components that can be reused by many different software projects lend themselves well to being sold on a market.[4] In other engineering industries, a component market enables buyers to choose from a wide variety of components that can be integrated into their products, making possible complex products that would have been hard for a single company to realize alone.

[3] Building complex software using only visual means has been largely discredited. Anders Heljsberg, former architect of the Borland Delphi development environment, discusses a purely visual development product from Borland that was canceled before it shipped (Microsoft 2005): “At that time there was all of this talk about Software ICs and plug-and-play[... You] have a visual designer and put down your components and then wire them together and it all sounded great[...]. It turned out [that] you couldn't build anything with this stuff [...]. When it comes to visual programming, a line of code is worth a thousand pictures, because you die a slow death in wires going from everywhere to everywhere.”

[4] Software components that are released under an open source license, and are thus typically available at no cost, are increasingly used in industry (Vigder 2001).

Some proponents of software components believe that the same kind of market is sustainable for software parts. Ivar Jacobson, one of the founders of the Unified Modeling Language (UML), writes the following in a foreword to a book on software components (Heineman and Councill 2001a):

> I dream that we will get a component marketplace where different players can work. Some will play the role of selling components; others will buy components. [...] In the coming years, we will be able to revolutionize the production of software and then bring my final dream to fruition [...].

1.3 Defining a software component

Software components are notoriously hard to define (Olsen 2006). Many concepts have been graced with the "component" designation over the years, and far from all are compatible with the view taken in this thesis. By contrast, the *object* concept is comparatively well-defined—*objects encapsulate state and behavior.*[5] There is sufficient consensus on this definition that it is enshrined in a general-purpose dictionary (Merriam-Webster 2009a):

> [An object is] a data structure in object-oriented programming that can contain functions as well as data, variables, and other data structures.

The same dictionary has only a generic definition of the word "component"—a component is simply "a constituent part," that is, something that is part of a greater whole (Merriam-Webster 2009b). Mathiassen et al. (2001), in a book on object-oriented analysis and design, offer a somewhat more stringent definition:

> [A component is] a collection of program parts that constitutes a whole and has well-defined responsibilities.

While this definition is compatible with the software components of this thesis, it is very broad. Applied to software, the algorithms listed in the algorithms section of Communications of the ACM in the 1960s would qualify. McIlroy (1969) emphatically rejected the notion that these algorithms could qualify as components.

Two works provide more precise definitions that are used in this thesis. Szyperski et al. (2002:41), whose book "Component Software: Beyond Object-Oriented Programming" is one of the more influential in the field and one which is cited frequently in this thesis, offer this definition:

> A software component is a unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to composition by third parties.

Heineman and Councill (2001a:7), whose book "Component-Based Software Engineering: Putting the Pieces Together" is a collection of articles related to this field, offer this definition:

[5] Beyond this fundamental accord, there is considerable room for diversity. In most traditions, objects are created from blueprints called classes, but in some, they are cloned from other objects serving as prototypes. Also, object-oriented languages differ in their degree of support for classes to inherit the implementation of other classes, and if so, whether they are substitutable for the classes they inherit from.

> [A software component is] a software element that conforms to a component model and can be independently deployed and composed without modification according to a composition standard.

These two definitions capture the essence of how software components should be used. They are light on technical details, though. For the purposes of this thesis, the following pragmatic definition is adopted in addition to the above definitions:

> A software component is a container of instantiable classes, as understood in the field of object-oriented programming. A component has an identity at runtime, but no observable state. Different versions of a component may co-exist at runtime.

In their books, both Szyperski et al. and Heineman and Councill make provisions for components that are not object-oriented. Szyperski et al., in particular, stress that a component may be implemented using a procedural or functional language. This thesis takes the position that, at least from the outside, a component is a container of classes—a class superstructure. Supporting this position, Aleksy et al. (2005:6) state that "component technology [...] is conceptually still built upon the ideas and technologies of object-orientation." This view does not preclude a component implementation from being written in a procedural language, or from using the services of a functional language, but from the outside, a software component is object-oriented.

An alternative view of the component concept is that a component is a stateful entity that is placed on a form in a visual designer, its behavior customized through visual means. A push button in those environments, for instance, may thus be referred to as a "component." Others hold that a component is little more than a class, or a class "plus stuff" (Venners and Eckel 2003). These views are not compatible with the view adopted by this thesis.

1.4 Fleshing out the definitions

"One thing can be stated with certainty: components are for composition." (Szyperski et al. 2002:3). By composition, Szyperski et al. refer to the process of assembling a system using a selection of compatible components. This is only possible if components have a shared understanding of the outside world, what it offers, and their responsibilities toward it. Such a set of standards is known as a *component model*, of which there are many incompatible models.[6] Implementations of component models—that is, the accompanying runtime software and tools—are sometimes known as "component frameworks." Without a component model implementation, components cannot be used.

Components have responsibilities to their environment, but also to other components and to the clients that make use of their services. A component must advertise its services somehow, and must let its clients know what it expects of them. This information is communicated using *interfaces* that may be realized by an arbitrary number of implementations that agree to abide by their rules, and should be seen as binding contracts. Classes, which are contained in components, are the implementation entities that realize interfaces by *implementing* them. An interface is a freestanding, independent entity that is neither part of the classes implementing it, nor part of the code that accesses functionality through it.

[6] There are often *bridging solutions* that help components written for one component model communicate with those written for an otherwise incompatible model (Weinreich and Sametinger 2001).

Incoming interfaces communicate what components are capable of, while *outgoing* interfaces communicate what is expected of clients that make use of their services. As components are encapsulated entities exposing no implementation details, interfaces are the sole means of accessing them. They consist of an arbitrary number of operations that for incoming interfaces correspond to actions that the component may undertake on behalf of a client. Operations in outgoing interfaces are called by components, typically to notify clients of events that occur. An operation is similar to a method in an object-oriented language, but additional constraints are often imposed by component models, related to the types used for operation arguments as well as error handling. Incoming and outgoing interfaces are also known as *provided* and *required* interfaces, respectively. They form the basis for *connection-oriented programming*, enabling components to be glued together.

When a statically linked procedural library is used, the implementation is known at compile-time *(a priori)*. Interfaces to such libraries are known as *direct interfaces*, as the implementation is known beforehand. Object interfaces are *indirect*, in the sense that the actual implementation is only known at runtime. This is also known as *dynamic dispatch* or *late binding* (explored in detail in Chapter 3). Dynamic dispatch enables classes to have multiple personalities by implementing multiple interfaces—this is known as *polymorphism* in object-oriented programming.

It is hard to overemphasize the role of interfaces—they make it possible to program to a specification, and not to a concrete implementation. Interfaces play the same role as the standards that allow electrical outlets and the plugs they accept to work together—whether a vacuum cleaner or a power tool is connected to an electrical outlet is immaterial, as long as it can draw current in a standardized way.[7] Two components that implement the same interfaces and exhibit the same runtime behavior are substitutable for one another (known as *the principle of substitutability*) (Eriksson et al. 2004). This is often cited as an important business case for component technology—enabling a company to replace a component with a less expensive one from a competing vendor, or replacing a component from a vendor that has gone out of business (Tracz 2001). (In practice, there are monumental challenges to overcome to make the behavior of all but the simplest components compatible to the degree required to make one component completely substitutable for another.)

A component is a self-standing, independent unit that may be deployed separately from the clients that use it. It has no observable state. The source code of a component may or may not be available; it may be shipped only in binary form. If a component is dependent on any entities other than the environment it is written for, including the component model, such dependencies should be explicitly noted in machine-readable form (Heineman and Councill 2001b). Components should be able to indicate their version, making it possible for a component model implementation to simultaneously load different versions of the same component. (In large systems with complex dependencies, it is common for components to indirectly depend on different versions of the same component.)

It must be possible to differentiate between different components, classes and interfaces, at compile-time and at runtime. For this reason, these entities are assigned names, one that is used at compile-time and one that is used at runtime. The compile-time name is for the benefit of human developers, and should thus be human-readable, but need not be unique.

[7] A well-worn joke is that we love standards, that is why we have so many of them. Obviously, if there is one "standard" per implementation, nothing has been gained in the way of interoperability, and the large number of standards for electrical outlets helps to illustrate this.

The name used at runtime, however, must have a very high probability of being globally unique. The runtime name assigned to a component or interface must be such that it is very probable that no such name has been created in the past, and that no such name will be created in the future. Component models use a variety of strategies to try to ensure that this property holds. Services are often provided that activate components and instantiate classes given their runtime names.

Component models standardize a host of other facets. These include error handling, memory management, the deployment file format for components and metadata (such as type information available at runtime).

In addition, component technology is also associated with a number of more complex concepts:

- **Programming language agnosticism.** Component models dictate standards that make it possible to access components from a variety of programming languages, and also write components using different languages. Interacting with a component written in a different programming language should ideally feel as natural as interacting with code written in the native language.

- **Location-transparent invocations.** Component technology may be used to enable a client to remain oblivious to whether a component is executing in the client's operating system process, in another process on the same computer, or on a different computer altogether, perhaps halfway across the world. Component model implementations can make all components appear as though they are executing in the client's process, regardless of whether this is actually the case. In an enterprise setting, this is one of the selling points of component technology, as it makes it possible for services running on different servers and on different platforms to interoperate (which is often known as *distributed computing*).

- **Declarative services.** Some component models support associating components and objects with declarative attributes that help programmers manage such disparate things as concurrency, database transactions and security boundaries (Szyperski et al. 2002:430). Declarative attributes save the programmer from having to write error-prone code, serve as machine-readable documentation and make it possible to enforce architectural rules.

1.5 Beyond object-orientation?

Object-oriented programming has been criticized for failing to enable code reuse, and as a result for failing to establish significant "object" markets (Sullivan 2001; Pfister and Szyperski 1996). Component technology is designed to overcome these perceived shortcomings, and is by its proponents considered to be not only an evolution of object-oriented programming, but a successor paradigm in the Kuhnian sense.[8] Heineman and Councill (2001c) elaborate:

> We believe that object-oriented [...] analysis, design, and programming were over-hyped and over-sold. Early courses and seminars in [object-orientation] described the technology as an elixir for all that was wrong with years of poor analyses,

[8] Thomas Kuhn argued that science does not progress linearly, but rather in transformational, revolutionary shifts (Chalmers 1999).

> design, and programming practices. It didn't take long for information technology [...] departments and many independent software vendors [...] to realize that [object-orientation] was a very expensive methodology that ironically had been sold to senior management as a way to save money.

In 1994, Jon Udell wrote of "object technology" in the imperfect tense:

> Object technology failed to deliver on the promise of reuse. [...] What role will object-oriented programming play in the component-software revolution that's now finally under way?

Indeed, distributing and selling classes in binary form has traditionally been complicated by the lack of standards—even different implementations of some languages, such as C++, are binary incompatible. Distributing classes as source code is often not desirable for vendors that wish to maintain some modicum of secrecy. Obviously, lack of interoperability hampers adoption. What component technology brings to the table is the promise of standards that make reuse possible across different programming languages and vendors (Sullivan 2001). Indeed, component models typically standardize objects in addition to components.

Object-orientation gives developers the tools to define entities that encapsulate and operate on their own state. Components provide a higher-level abstraction, much like a library or a module, as the containers of classes. Hence, object-orientation and "component-orientation" are orthogonal and complementary concepts.

1.6 Muddying the waters

Having disparate services residing on different machines and implemented in different programming languages is not the sole domain of component technology. Servers and clients have communicated using well-specified protocols long before the advent of component technology.[9] Nor is reuse an innovation introduced with component technology. Desktop applications have successfully reused functionality from class libraries, procedural libraries and the operating system on which they run for years. Thousands of dynamically linked libraries have been developed and sold, and nary a software component, as defined on page 5, has been in use. Component technology is at its most pure an attempt to standardize some of the infrastructure that has been continually reinvented over the years, an ambitious attempt to create standards for everything from what an object looks like in memory, to the protocol used when two objects communicate over a network, to the declarative attributes used to safely store data in a database. Component technology as a term is thus nearly all-encompassing, and some implementations are as a result very complex.

The definitions adopted for this thesis deliberately exclude many successful forms of software reuse. In a sense, the operating system and its applications make up the most successful component ecosystem in existence: standardized components in the form of processes that communicate with the component model implementation—the operating system—using well-defined standards—system calls (Weinreich and Sametinger 2001). Dynamically linked procedural libraries are a very successful form of reuse, and provide robust language independence by standardizing procedural invocations (Szyperski et al. 2002:209).

[9] Indeed, component technology's support for location-transparent invocations grew out of traditional remote procedure calls.

Despite their successes, none of the aforementioned entities qualify as components under the definitions adopted at the beginning of this chapter. Applications are too coarse-grained and too project-specific to be considered components—applications are built by composing components, they are not themselves components. Dynamically linked libraries provide a useful platform on which to build component technology (as detailed in Chapter 4), but cannot in themselves be considered components, as they provide no vendor-agnostic support for objects, and different versions of the same library cannot easily be loaded simultaneously, as they have no identity at runtime.[10] Component technology lays down the standards that enable the properties mentioned in this chapter, including rich reuse, support for object-orientation, language- and platform-neutrality, location-transparent invocations and declarative services.

1.7 Enterprise services

A number of enterprise services are often layered on top of component models. These services automate some of the tasks people often face in enterprise settings, typically by making it possible to replace manually written, and often error-prone, imperative code with declarative attributes. Enterprise computing often implies that services run on different machines, which benefits from the support for location-transparent invocations often provided by component models.

Component model implementations used in the enterprise typically come bundled with standardized servers that host components, known as *application servers*. Having a standard server implementation helps developers focus on writing domain-specific code. An application server is used to encapsulate *business logic*, and typically sits between a user-facing front-end, such as a web page or a standard application, and a persistence layer in the form of a database server. An application server is a complete server, but needs business logic in the form of components to be plugged in before it can do anything useful.

This is a list of some enterprise services typically provided by component models intended for use in the enterprise:

- **Transaction processing.** Modern database servers use transactions to group a set of database operations that must be either committed as a group, or not at all, in order to maintain data integrity. Component models automatically manage transactions if declarative attributes to that effect are set. A transaction is committed if a set of operations complete successfully, and is rolled back if errors are signaled.

- **Loosely coupled events.** Traditionally, event consumers subscribe to events by communicating directly with the event producer. This feature decouples consumers and producers by introducing an event service, which enables features such as efficient multicasting and filtering. Events can often be subscribed to by setting a declarative attribute.

- **Deferred processing.** Some requests need not be performed right away and can be postponed for later. If the component which is to perform the service is not available,

[10] A dynamically linked library can certainly have an identity at runtime, if loaded explicitly and not implicitly. Combining explicit loading with a way to retrieve the version number from a library, multiple versions of a library can be loaded simultaneously. Adding object semantics on top of procedural invocations is also possible, and with these additions, a dynamically linked library would qualify as a software component. These issues are discussed in Chapter 4.

the request is put in a queue. A component may be unavailable for a variety of reasons, such as when the network is unreachable or the computer on which the component runs is powered off. This feature may also be used to schedule jobs to execute during off-peak periods, and to balance the load across several machines.

- **Security.** Certain services may only be available to components running as privileged users. The security model of a component model can help enforce security policies by preventing access to certain services. Access may be barred to certain components, implementations of certain interfaces, or even to specific operations.
- **Just-in-time activation.** Component models normally strive to keep instances alive for as long as there are active references to them. This feature helps conserve resources on a server by destroying instances that are referenced, and transparently instantiating them anew when they are accessed.
- **Synchronization.** Objects that are not reentrant should not allow multiple simultaneous threads of execution. Such objects can be tagged with a declarative attribute that enables a component model implementation to bar concurrent access.
- **Object pooling.** Keeping instantiated objects in a pool for later reuse can significantly boost performance. A pool contains instantiated objects, ready to be used. When an active object is no longer used, it can be put back into the pool for later use. Declarative attributes can typically be used to configure certain aspects of the pool, such as its maximum size.

CHAPTER 2

Realizing software components

A large number of component models partly or fully conform to the definitions of Chapter 1. Some are specific to a certain domain, while others are a more generic nature. Domain-specific component models include plug-ins for software applications (such as web browsers, photo manipulation software and audio players), as well as components running in application servers (for instance, generating personalized web pages on-demand with the aid of a database server). This thesis is mainly concerned with attempts at building component models that are not limited to any one domain.

This chapter serves the dual purposes of tracing the evolution of component models, and in so doing, introduces many of the concepts that enable software to be usable as components. The large number of component models with differing goals and terminology makes this an imprecise task.

Literary criticism cautions that *there is no final reading of a text.* Following this, and considering component technology in all its forms as the text, no one interpretation can be seen to represent the objective “truth.” Yet, this chapter attempts to categorize component models into three somewhat arbitrary generations. The first-generation component models are proprietary, whereas the second-generation component models are standards-based. Both target native code. The third-generation component models discussed here target virtual machines.

2.1 First-generation component models

Components written for first-generation component models are often used in a visual designer that helps developers rapidly construct graphical applications. These component models focus solely on the desktop and not on the enterprise. Such components typically contain graphical *controls* (also known as *widgets*), that are manipulated visually. A designer can drag a control to an application window that is under construction, position and resize it, and customize it by changing its *properties.* A push button may allow its appearance to be customized, for example. A programmer is likely to want custom code to be executed when certain actions occur at runtime, such as a user pressing a button, which is accomplished by adding *event handlers* to controls.

Components that contain non-graphical controls may also be created for these environments, in some cases significantly extending their functionality. Such an environment may only allow slow-performing scripts to run, making it impractical or impossible to write lower-level code like parts of a network protocol stack. Higher-performing code, written using a traditional programming language, may be packaged as components, whose non-graphical controls are then composed using a visual designer. Provided that all required components are available and can be used as-is, this approach makes it easy to combine components in ever-new assemblies. Creating graphical front-ends to corporate relational databases is a common use of application builders that use first-generation component models (making use of data-aware controls).

Components written for a first-generation component model are, at least in theory, tied to a single environment, and follow only vendor-specific standards.[1] They run in the caller's operating system process, do not necessarily rely on interfaces and dynamic dispatch, use explicit memory management and are written for the target machine's native instruction set. Many of the properties enumerated in Chapter 1 do no apply to these components. Components are either distributed as dynamically linked libraries that export vendor-mandated symbols, or are simply statically linked with the applications that use them. Object semantics, if any, are proprietary to the vendor. Examples of environments supporting these component models include early versions of Microsoft's Visual Basic and Embarcadero's Delphi.

Despite their simplicity—or perhaps because of it—components written for these environments have done well, both as vehicles for reuse and in the marketplace (Udell 1994).

2.2 Second-generation component models

The component models introduced after the first generation are far more ambitious, and encompass far more functionality. No longer tied to a single platform or programming language, they are often based on published specifications or even formal standards. They make it possible to use services running in other processes and on other computers, and include standards for such disparate things as dynamic dispatch, memory management, distribution formats and naming. They target the enterprise in addition to the desktop. They abstract away notions of where a component is executing, and tackle issues ranging from concurrency to working with database transactions. Their ambition, compounded by the fact that they are layered on top of traditional programming languages and target a wide range of machines, makes them significantly more complex than the comparatively simple first-generation component models. As they target platforms with no object models of their own, they standardize objects in addition to components. Microsoft's Component Object Model (COM) and the Object Management Group's Common Object Request Broker Architecture (OMG's CORBA) are prominent members of this generation.

2.2.1 Realizing interfaces

Objects provided by a component conforming to a second-generation component model are never accessed directly, they are always accessed through one of the interfaces they implement. As components may be written in languages with no native support for interfaces, they are defined in an auxiliary descriptive language that allows no code, only definitions of interfaces,

[1] The dearth of published vendor-neutral standards does not preclude a vendor from reverse-engineering the products of a competitor, and offering support for hosting the same kind of components (Udell 1994).

their operations and the types that are used. These languages are broadly known as *interface description languages*, widely known by the abbreviation IDL. (A large number of incompatible languages lay claim to the name "IDL.")

Code generators that come with component model implementations, known as *IDL compilers*, take an IDL file as input and generate *language bindings* for a supported programming language.[2] Language bindings make it possible to access components from multiple languages and ideally conform to the target language to a great extent, making components feel like a natural part of the target environment. Some language bindings, typically for scripting languages, may not allow components to be authored in the target language. Such languages are restricted to accessing components written in other languages.

An operation in an interface is mapped to the most appropriate construct in the target language. For a procedural language, this is a procedure or a function; for an object-oriented language, this is a method or an operation. A type is mapped to the native type most appropriate for the target language—a "number" type used in an IDL file may be mapped to the "double" type in a binding for the C language, for instance. If the execution of an operation fails, this is normally communicated by throwing an exception in a target language that supports exceptions, or by returning an error code in a language with no such support. (True return values are communicated using output arguments when the native return value is used for error information.)

2.2.2 Calling in-process components

Invoking operations of components that run in-process can be realized in two primary ways. One approach is to lay down a binary standard that covers all components written for the component model, regardless of what programming language they are implemented in. At its core, this approach is similar to how procedural libraries work. A procedural library can be used by a program written in a programming language different from that of the library, as long as the program and library are both compiled for the same architecture and the program is aware of how to call procedures contained in the library. Procedural invocations are covered by standards known as *calling conventions* that standardize the mundane technical means used when one procedure calls another. This includes the order in which arguments are transferred, which arguments are transferred on the call stack and which are transferred in CPU registers (if any), handling of return values and stack clean-up.

Object-oriented programming languages need to handle all of the above, but also need to take objects into account. Specifically, the instance data of the receiving object (its state) must be passed, and there must be a way to realize dynamic dispatch (that is, to look up the implementation at runtime as opposed to at compile-time). Second-generation component models target architectures that are not, in themselves, object-aware, and thus must standardize object invocations (they also act as object models). These issues are explored in Chapter 3 and Chapter 4. COM is likely the best-known component model that is a binary standard.

The alternative approach, exemplified by CORBA, is to standardize on the IDL dialect in conjunction with formalized language bindings. This approach requires clients to rely on runtime software to make calls to objects. This software is supplied with the component model implementation and, at least partly, runs in the same process as the client. Vendors of different component model implementations may use different means to realize component calls.

[2] Language bindings can also be generated from other representations of interfaces, such as COM type libraries, which contain roughly the same information as IDL files in a format that is more easily parsed.

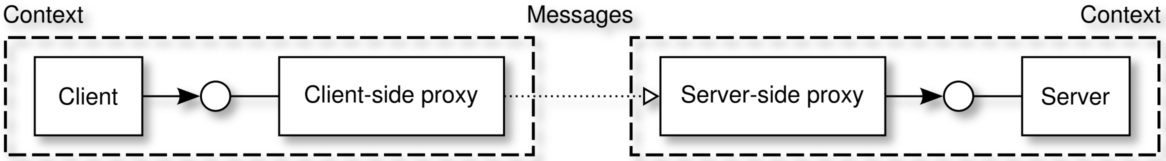

Figure 2.1 Client-side and server-side proxies

In order to ensure that a client can use any conforming component model implementation with components tested with a different implementation, different language bindings must be carefully standardized. A client written in one programming language can communicate with a component written in another language as long as the component model implementation supports both languages. The runtime software supplied with the component model implementation is responsible for translating calls from one language to another.

2.2.3 Calling out-of-process components

The raison d'être of many component models, such as CORBA, is to enable distributed systems in the enterprise. These systems typically consist of many disparate services that run either on different computers[3] or in different operating system processes on the same machine. Some of these services may be written explicitly for the component model, while others may consist of legacy software wrapped by components (acting as adapters, in the design pattern sense (Gamma et al. 1995)). Distributed architectures in this fashion have many names, including Service-Oriented Architecture (SOA) and Service Component Architecture (SCA).

Component models strive to make it easy to call remote objects, generally by providing a stand-in object that runs in the client's context and forwards all calls to the remote object. The stand-in object masquerades as the remote object by implementing the interface that the client wishes to use to communicate with the remote object. A stand-in object is known as a *client-side proxy*. On the server end, a *server-side proxy* receives messages sent by the client-side proxy, and dispatches calls to the server-side object using standard calling conventions. This is illustrated in Figure 2.1. The classes that are instantiated to form proxies are often generated by an IDL compiler. It is said that component technology enables location-transparent invocations due to this ease-of-use property. Using component technology to call out-of-process components is one way of realizing *inter-process communication* (IPC) and *inter-machine communication*. This is similar to traditional remote procedure calls, but with object semantics.

Component models that are binary standards, such as COM, have no need for proxies when two objects communicate that run in the same process; they communicate directly.[4] Without such a binary standard, as with CORBA, objects rely on a runtime part of the component model implementation to facilitate in-process calls, hence requiring the use of proxies for all types of calls. A client-side proxy masquerades as the target object and forwards calls to the runtime system, which then forwards the call to the server-side proxy, which finally calls the target object (which is not necessarily remote).

[3] Increasingly, many of these "computers" are virtualized, and thus run on the same physical computer.

[4] COM sometimes does use proxies even for objects that run in the same process. See Chapter 7.

When making an in-process call, execution leaves the client operation body for the duration of the call. This behavior does not occur naturally when calling on the services of a remote object—if no steps are taken to prevent this behavior, execution only leaves the client operation body while a proxy forwards the call to the remote object. To prevent the client process from continuing to execute while the remote object is processing a call, the client process or thread is blocked. When it resumes execution, the remote object has finished execution, and the return value (if any) of the invoked operation is delivered to the client, just as though the target object had been running in-process. This type of call is known as a *synchronous* call.

Synchronous calls are convenient, as they ostensibly work just like calls to objects running in-process.[5] The component model implementation takes care to hide the technical machinery used to communicate with the remote object. There are times when it is desirable for the client code to continue execution while a remote object is executing. Calls enabling this behavior are known as *asynchronous* calls. The client and remote object thus execute concurrently, and the remote object notifies the client when it has finished executing, if a reply has been requested.[6] This is beneficial for long-running operations, but is slightly less convenient for developers, as clients need to keep track of additional states (whether or not it is waiting for a remote operation to complete). With synchronous calls, the client is relieved from having to keep track of these states, as it is blocked from executing while the remote call is on-going. Some component models only support synchronous calls. Those that do support asynchronous calls often default to synchronous calls.

In order to call a remote object, a component model implementation must *marshal* calls across processes and, possibly, machines. Marshalling entails converting an invocation into an unambiguous binary format, the *wire format*, which holds information on the operation to be called, as well as its arguments. Component models thus standardize network protocols used to carry calls to out-of-process objects. Client-side proxies are responsible for creating a data stream that conforms to this protocol and sending it to a server-side proxy, which *unmarshals* the call and invokes the corresponding operation on the remote object. If the client asked to be notified upon completion of the remote operation, the server-side proxy packages the return value in a data stream that conforms to the protocol, and sends it to the client-side proxy which returns it to the original caller. The client and the remote object are both unaware of the work done by the server-side and client-side proxies to facilitate the call.

Simple types such as strings and integers are always passed by value when marshalled, in effect placing the argument data in the data stream. References to an implementation of an interface, however, are often passed by reference, meaning that only a means of identifying the implementation is placed in the data stream. As noted, in order to communicate with a remote object, client-side and server-side proxies need to be set up. When a remote object receives an interface argument that refers to an object that runs in the client's context, a reverse connection needs to be set up—in order to communicate with the object that runs in the client's context, the remote object needs to have a client-side proxy in its context that

[5] There are subtle problems associated with synchronous calls. Notably, they may cause *deadlocks* (permanently blocked processes or threads), and they are inherently much slower than the in-process calls they try to mimic. These issues are briefly discussed on page 100.

[6] The reply will generally be delivered by sending a message to the client. Long-running programs that sit idle from time to time, such as servers and applications that interact with users, are generally message-oriented. The standard implementation of such programs is to have an outer message loop that waits for messages and dispatches them to the target objects. The message loop is often part of a system library, and is thus often hidden from developers.

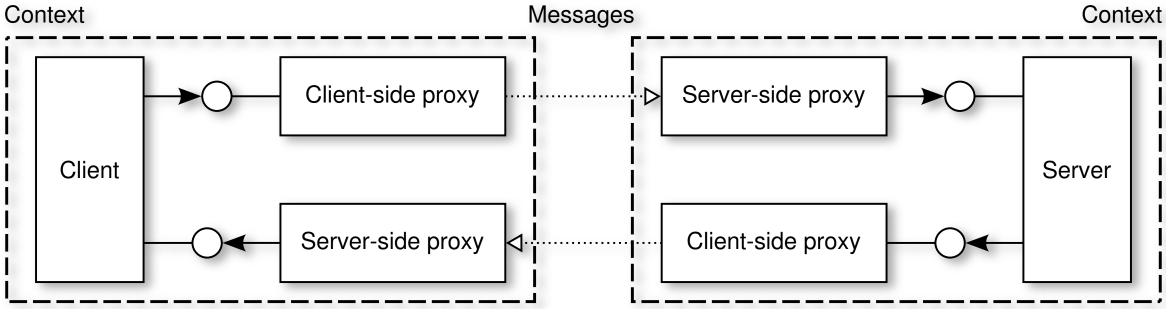

Figure 2.2 Interface reference passed to a server

communicates with a server-side proxy that runs in the client's context (see Figure 2.2).

Some component models support passing interface references by value. The instance data of an object that supports being passed by value is written to the data stream (it is *serialized*) by the client-side proxy, and is reconstructed *(deserialized)* by the server-side proxy.

The network protocols used by component models should generally be well-specified in terms of how data is represented—for instance, different machine architectures often use different byte orders. A component model that does not allow calls across a network, and is thus limited to one machine, may relax these rules, as the client-side and server-side proxies are guaranteed to run on the same machine (and have likely been generated by the same IDL compiler).

2.2.4 Late binding versus very late binding

The implementations of classes are, as noted, always hidden behind their interfaces. As such, a client is unaware of the precise implementation it is calling, and thus cannot be bound to an implementation at compile-time. Retrieving a reference to the implementation is done at runtime. The client does, however, have compile-time knowledge of the layout of the interface—it knows, at compile-time, what operations are available, their arguments and their types. It thus binds to the implementation of the interface dynamically, but checks the validity of calls statically. Again, this is known as dynamic dispatch, or late binding.

In some environments, it may not be possible to have compile-time knowledge of interfaces and their operations. An interpreter for a scripting language, to mention the most prominent example, is not compiled against the myriad of components that a script may want to access. As such, the validity of calls must be checked at runtime. This requires that metadata in the form of runtime type information is not only available at compile-time, but also at runtime. This is known as *very late binding.*

Some component models, notably COM, take the approach of supplying a standard, domain-agnostic interface that is used for dynamically checked invocations (in COM, this interface is named `IDispatch`). Classes that wish to support being invoked by environments that need to check invocations at runtime must also implement this interface (often in addition to interfaces to which calls are checked statically). Script interpreters that support such a component model have compile-time knowledge of this interface, and facilitate access to it from scripts. The implementation of this interface can use self-contained code specific to a certain class to check invocations, or the implementation can be generic and rely on runtime-accessible type information deployed to end-users' systems (discussed on page 73). Other component models, notably CORBA, require clients to use a single system-provided, domain-agnostic implementation to invoke calls using very late binding (which also requires the presence of type information at runtime).

2.2.5 Managing memory

As components may be written in languages that use different memory management strategies, it is impossible for the runtime system of a component model implementation to allocate memory for objects. Component models that use formalized language bindings delegate object instantiation to them, whereas those with no such formalized bindings generally require that components provide *factories* that instantiate objects. In order to instantiate objects, component model implementations must be able to map runtime names of objects to the components that house them, and to their factories.

Some second-generation component models set standards for managing the lifetime of objects. First-generation component models use explicit lifetime management, meaning that an instance can be explicitly destroyed by any party. This strategy does not work well with a large number of independent clients that are not aware of one another. A client can only be expected to know when the client itself no longer has a need for an object reference; it cannot be expected to know, and should not know, whether other clients still have live references to an object. To solve this, many second-generation component models use the simple strategy of having every object include a count of the number of clients that have live references to it—this strategy is known as *reference counting*. Clients notify objects when they store a reference for later use (thus incrementing the reference count) and when they no longer have a need for a previously-held reference (thus decrementing the reference count). The object destroys itself when there are no clients that hold references to it (that is, when the reference count reaches zero). There is thus no need for a destruction counterpart to factories, as objects are responsible for managing their own lifetime.

Reference counting is a form of cooperative garbage collection. It is simple to implement and understand, but has the significant weakness that all clients need to play by the rules at all times. If one client malfunctions, memory leaks or system breakage will ensue.

Counting references has the well-known problem of not being able to handle cyclic references. Consider a trivial example: If an object has a reference to another object, which itself references the first object, these two objects will never be destroyed, regardless of whether there are external parties that reference either; they keep each other alive. (Non-trivial examples normally include a much larger set of objects in the cycle.) This scenario is especially common when one object wishes to subscribe to events from another—the subscribing object obviously must reference the event-providing object in order to subscribe, and when the subscription request has completed, the reverse is also true. Second-generation component models often include means—often error-prone—that seek to make it possible to break such cycles (Szyperski et al. 2002:335).

2.3 Third-generation component models

The complexity of second-generation component models stems partly from their ambition, and partly from the fact that they only introduce new technology on top of traditional technology, making no attempt to redefine the traditional technology and the abstractions that come with it. These component models work with traditional languages, are written for traditional platforms, and provide tools that are mere adjuncts to the tools that come with the platforms they target, such as compilers and linkers. The old technology is left undisturbed—it is simply built on top of. Added to, not changed.

Third-generation component models are radically different in that they no longer target a large number of platforms. They target just one, and one that already provides many of the services associated with second-generation component models, making for conceptually much simpler technology (but no less powerful). Prominent examples of the platforms that third-generation component models are built on top of include Sun's Java platform, and Microsoft's .NET platform. Concrete component models that take advantage of these platforms are studied in Chapter 5.

The platforms that third-generation component models are built on have virtual machines at their core. As a result, components written for such a platform run on many different native platforms, but only need to target one virtual platform. As long as a component does not make direct use of the services offered by the native platform, it will run anywhere there is an implementation of the virtual machine. Technologies such as *just-in-time compilation* (JIT) and *ahead-of-time compilation* (AOT) help make code written for virtual machines competitive with code compiled for native architectures from a performance point of view.[7]

There have been proposals suggesting that components should never run in the caller's operating system process to protect stateful objects instantiated by the component from tampering, using hardware memory protection available in modern architectures (Szyperski et al. 2002:79). Such heavy-handed methods are made obsolete by forcing components to be written for a virtual machine that has no instructions that allow code to access memory directly (or relegates such instructions to an "unsafe" mode that must be specifically enabled).[8,9]

Second-generation component models are forced to standardize concepts not directly related to software components, such as memory management. The platforms that third-generation component models build upon have subsumed many of these technologies, making it possible for third-generation component models to focus on enabling software components. This makes it significantly easier to use and understand third-generation component models, as they simply extend the rich functionality of the platforms on which they are built.

[7] Just-in-time compilation and ahead-of-time compilation compile the machine instructions for the virtual machine (known as *bytecode*) to machine instructions for the native processor, essentially treating the bytecode the same way a compiler back-end treats the intermediate language produced by its front-end. Ahead-of-time compilation, also known as "static compilation," compiles bytecode to native machine code before the program is executed. Just-in-time compilation, also known as "dynamic compilation," compiles the whole program or parts of it at runtime (parts that are not compiled are interpreted). Just-in-time compilation is interesting in that it can take runtime aspects into account, meaning that the code it generates can, in theory, be better than statically compiled code—eliminating unneeded dynamic dispatch or simplifying complex code that is always used in one specific way, for instance (Stoodley et al. 2007).

[8] Some instructions may only be safe if used as prescribed. For instance, the Java virtual machine does not allow programs written for it to jump to locations that are outside the currently executing method. A "bytecode verifier" can be used to verify that untrusted code does not flout these rules before it is executed.

[9] This might be considered a computer application of the Sapir-Whorf hypothesis in linguistics, which postulates that natural language shapes our perception of the world (McGee and Warms 2004). George Orwell used the Sapir-Whorf hypothesis to great effect in his dystopian novel Nineteen Eighty-Four (2003), in which the dictatorship has constructed a new language, *Newspeak*, which is a variant of English lacking words representing concepts that the regime would prefer the people did not ponder, such as "rebellion" and "freedom." A virtual machine with no instructions for memory freedom (in the sense of being able to access and write to arbitrary memory addresses) has no means of even expressing this concept.

These are some of the technologies standardized by second-generation component models that are part of the platforms used by third-generation component models:

- **Object model.** Interfaces, classes and objects are directly supported by the platforms discussed here, and by most of the programming languages that target these machines. Late binding, implementing interfaces, instantiating objects and even extending classes are thus directly supported by the virtual machines. Language-level support for interfaces means that no separate interface description languages are needed; interfaces are simply specified directly in the implementation language.
- **Runtime metadata.** These platforms maintain runtime-accessible metadata on classes, interfaces and other types, making it possible to explore these aspects at runtime. This support makes it easy to support very late binding, and thus scripting languages. Access to type information at runtime also makes it possible to forego the generation of proxy classes at build-time.
- **Memory management.** Third-generation component models rely on the automatic memory management used by their platforms, which is universally garbage collection.
- **Error handling.** Whereas some second-generation component models resort to using the return value of operations for error information, third-generation component models use the exception handling standardized by their platforms.
- **Naming.** The platforms supporting third-generation component models mandate their own naming scheme, which is simply adopted by the component models themselves.

Layering component technology on top of mature environments that already provide support for object-orientation, runtime type information, garbage collection and the like means that component technology can concern itself with only realizing software components, considerably simplifying third-generation component models.

CHAPTER 3

Demystifying dynamic dispatch

Dynamic dispatch is at the heart of object-based component technology. Without it, freestanding interfaces would not be possible. Dynamic dispatch, also known as late binding, makes it possible to program to a specification and bind to an implementation at runtime.[1]

Chapter 2 introduces some of the technology used to make component technology a reality. It glosses over the technical minutiae of dynamic dispatch, though, which this chapter aims to rectify. During the course of this chapter, two example programs, written in the C programming language, are developed that establish a set of conventions used when implementing object-oriented programs in C. These conventions essentially form a complete object model, supporting encapsulation and polymorphism, but not implementation inheritance or multiple interface inheritance. Chapter 4 builds on this material to sketch a component model.

The first example program, which does not use dynamic dispatch, is intended to set the stage for the second example program by introducing some fundamental concepts. The second example program builds on this foundation by introducing dynamic dispatch. C is a good choice for this task, as it has all the features needed to implement dynamic dispatch on top of an existing procedural language. The first example also appears as functionally equivalent Java code. Java already has the language constructs which are implemented for C in this chapter—classes, objects and interfaces.

There have been many attempts to bring object-oriented programming to C. In an article from 1997, Samek presents a battery of C macros and best practices making it possible to implement object-oriented constructs, including implementation inheritance. Bjarne Stroustrup's first C++ compiler, Cfront, produced C code to be compiled by a traditional C compiler (Stroustrup 1994:66).

Another example is the GNOME desktop environment—primarily used with Unix-like operating systems—which uses the GObject object model pervasively. GObject is written in C, and may be used directly in native applications or through bindings to other languages. Vala is a fully object-oriented language that targets native code and uses GObject as its object model. Like Cfront, its compiler produces C code.

[1] The form of dynamic dispatch discussed in this thesis is *single dispatch*, as opposed to *multiple dispatch*. Single dispatch resolves what implementation to use by only taking into account the type of the first argument (which by convention is the instance data of objects in object-oriented systems). Multiple dispatch takes the types of all arguments into account.

Some of the examples in this chapter are heavily abridged due to space constraints. The reader is encouraged to download and experiment with the source code. The C source code should be compatible with all compilers adhering to the C99 standard.[2] It is available at `http://www.polberger.se/components/`.

3.1 A binary tree node in C

A node in a binary tree has at most two children, referred to as the *left* and *right* nodes. In object-oriented programming, a tree node is naturally modeled as an object.

Implementing such a node in Java is straight-forward, as the Java language and platform come with support for object-oriented programming, automatic memory management and structured error handling (in the form of exceptions). A concise sample implementation, which allows arbitrary data to be associated with a node, is shown in Listing 3.1.

Listing 3.1 BinaryTreeNode.java

```
/**
 * Instances of this class represent immutable nodes in a binary
 * tree. Arbitrary data may be associated with a node.
 *
 * @param <D>
 *    the type of data objects maintained by this node and its
 *    children.
 */
public final class BinaryTreeNode<D>
{
  /**
   * The client-defined data associated with this node, or {@code
   * null} if there is no such data.
   */
  private final D data;

  /**
   * The left node of this node, or {@code null} if there is no such
   * node.
   */
  private final BinaryTreeNode<D> leftNode;

  /**
   * The right node of this node, or {@code null} if there is no such
   * node.
   */
  private final BinaryTreeNode<D> rightNode;

  /**
   * Creates an instance of this class.
   *
```

[2] The Java source code has been tested with Sun Microsystems's Java Development Kit (JDK) for the Standard Edition, version 6. The C source code has been tested with the C front-end of the GNU Compiler Collection (GCC), version 4.3.3. The example programs have been developed on an x86-64 Linux system, and Valgrind 3.4.1 has been used to check the C programs for memory leaks.

```
   * @param data
   *   the client-defined data of this node, or {@code null}.
   * @param leftNode
   *   the left node of this node, or {@code null}.
   * @param rightNode
   *   the right node of this node, or {@code null}.
   */
  public BinaryTreeNode(D data,
                        BinaryTreeNode<D> leftNode,
                        BinaryTreeNode<D> rightNode)
  {
    this.data = data;
    this.leftNode = leftNode;
    this.rightNode = rightNode;
  }

  /**
   * Returns the client-defined data associated with this node, or
   * {@code null} if there is no such data.
   *
   * @return
   *   the data associated with this node, or {@code null}.
   */
  public D data()
  {
    return data;
  }

  /**
   * Returns the left node of this node, or {@code null} if there is
   * no such node.
   *
   * @return
   *   the left node of this node, or {@code null}.
   */
  public BinaryTreeNode<D> leftNode()
  {
    return leftNode;
  }

  /**
   * Returns the right node of this node, or {@code null} if there is
   * no such node.
   *
   * @return
   *   the right node of this node, or {@code null}.
   */
  public BinaryTreeNode<D> rightNode()
  {
    return rightNode;
  }
}
```

Implementing the same class in C is more involved, as the object-oriented facets need to be handled explicitly. One facet that is not relevant to this example, however, is dynamic dispatch. As the `BinaryTreeNode` class does not implement any interfaces, it must be referenced directly (through its class interface). Also, as it is declared `final`, it may not be extended, and as a result, there is no ambiguity as to what implementation is referred to when the `BinaryTreeNode` type is referenced. This lack of polymorphic behavior allows the compiler to bind a client statically to the implementation.

Listing 3.2 shows the header file for a C implementation of this class, and Listing 3.3 the implementation.

Listing 3.2 BinaryTreeNode.h

```c
/**
 * @file
 *
 * This file contains function prototypes for the operations of the
 * <code>BinaryTreeNode</code> class.
 */

#ifndef INCLUSION_GUARD_BINARY_TREE_NODE
#define INCLUSION_GUARD_BINARY_TREE_NODE

#include <stdbool.h>

/**
 * This type represents the <code>BinaryTreeNode</code>
 * class. Instances of this class represent immutable nodes in a
 * binary tree. Arbitrary data may be associated with a node.
 */
typedef struct BinaryTreeNode_s BinaryTreeNode_t;

/**
 * Creates an instance of the <code>BinaryTreeNode</code> class.
 *
 * @param[in] pData
 *   the client-defined data of this node, or <code>NULL</code>.
 * @param[in] pLeftNode
 *   the left node of this node, or <code>NULL</code>.
 * @param[in] pRightNode
 *   the right node of this node, or <code>NULL</code>.
 * @param[out] ppThis
 *   a pointer to the variable which shall hold the instance of the
 *   <code>BinaryTreeNode</code> class. Must not be <code>NULL</code>.
 * @return
 *   <code>true</code> if the operation completes successfully,
 *   <code>false</code> otherwise.
 */
bool BinaryTreeNode_Create(void* pData,
                           BinaryTreeNode_t* pLeftNode,
                           BinaryTreeNode_t* pRightNode,
                           BinaryTreeNode_t** ppThis);
```

```
/**
 * Destroys this instance of the <code>BinaryTreeNode</code>
 * class. The children of this node will not be recursively destroyed
 * as a result of calling this operation. If the client-defined data
 * associated with this node has been initialized to point to memory
 * allocated at runtime, this memory must be manually deallocated
 * before calling this operation. This operation always succeeds.
 *
 * @param[in] pThis
 *   an instance of the <code>BinaryTreeNode</code> class. May be
 *   <code>NULL</code>.
 */
void BinaryTreeNode_Destroy(BinaryTreeNode_t* pThis);

/**
 * Returns the client-defined data associated with this node, or
 * <code>NULL</code> if there is no such data.
 *
 * @param[in] pThis
 *   a pointer to the instance data of this
 *   <code>BinaryTreeNode</code> object. Must not be
 *   <code>NULL</code>.
 * @param[out] ppData
 *   a pointer to the variable which shall hold the client-defined
 *   data. Must not be <code>NULL</code>.
 * @return
 *   <code>true</code> if the operation completes successfully,
 *   <code>false</code> otherwise.
 */
bool BinaryTreeNode_Data(BinaryTreeNode_t* pThis,
                         void** ppData);

/**
 * Returns the left node of this node, or <code>NULL</code> if there
 * is no such node.
 *
 * @param[in] pThis
 *   a pointer to the instance data of this
 *   <code>BinaryTreeNode</code> object. Must not be
 *   <code>NULL</code>.
 * @param[out] ppLeftNode
 *   a pointer to the variable which shall hold the reference to the
 *   left node. Must not be <code>NULL</code>.
 * @return
 *   <code>true</code> if the operation completes successfully,
 *   <code>false</code> otherwise.
 */
bool BinaryTreeNode_LeftNode(BinaryTreeNode_t* pThis,
                             BinaryTreeNode_t** ppLeftNode);

/**
 * Returns the right node of this node, or <code>NULL</code> if there
 * is no such node.
 *
 * @param[in] pThis
 *   a pointer to the instance data of this
 *   <code>BinaryTreeNode</code> object. Must not be
 *   <code>NULL</code>.
 * @param[out] ppRightNode
 *   a pointer to the variable which shall hold the reference to the
 *   right node. Must not be <code>NULL</code>.
 * @return
 *   <code>true</code> if the operation completes successfully,
 *   <code>false</code> otherwise.
 */
bool BinaryTreeNode_RightNode(BinaryTreeNode_t* pThis,
                              BinaryTreeNode_t** ppRightNode);

#endif // INCLUSION_GUARD_BINARY_TREE_NODE
```

Listing 3.3 BinaryTreeNode.c

```
/**
 * @file
 *
 * This file contains a class, <code>BinaryTreeNode</code>, instances
 * of which represent immutable nodes in a binary tree. Arbitrary data
 * may be associated with a node.
 */

#include <stdlib.h>
#include <stdbool.h>
#include "BinaryTreeNode.h"

/**
 * This structure represents the instance data of
 * <code>BinaryTreeNode</code> objects.
 */
struct BinaryTreeNode_s
{
  /**
   * Arbitrary data associated with this instance. May be
   * <code>NULL</code>.
   */
  void* pData;

  /**
   * A pointer to the left node pointed to by this node. May be
   * <code>NULL</code>.
   */
  BinaryTreeNode_t* pLeftNode;

  /**
   * A pointer to the right node pointed to by this node. May be
   * <code>NULL</code>.
   */
  BinaryTreeNode_t* pRightNode;
};

bool BinaryTreeNode_Create(void* pData,
                           BinaryTreeNode_t* pLeftNode,
                           BinaryTreeNode_t* pRightNode,
                           BinaryTreeNode_t** ppThis)
{
  BinaryTreeNode_t* pBinaryTreeNodeInstance = NULL;
  bool result = ppThis != NULL;

  if (result)
  {
    pBinaryTreeNodeInstance = malloc(sizeof(BinaryTreeNode_t));
    result = pBinaryTreeNodeInstance != NULL;
  }

  if (result)
  {
    pBinaryTreeNodeInstance->pData = pData;
    pBinaryTreeNodeInstance->pLeftNode = pLeftNode;
    pBinaryTreeNodeInstance->pRightNode = pRightNode;
    *ppThis = pBinaryTreeNodeInstance;
  }
  else if (ppThis != NULL)
  {
    *ppThis = NULL;
  }

  return result;
}

void BinaryTreeNode_Destroy(BinaryTreeNode_t* pThis)
{
  if (pThis != NULL)
  {
    free(pThis);
  }
}

bool BinaryTreeNode_Data(BinaryTreeNode_t* pThis,
                         void** ppData)
{
  bool result = (pThis != NULL) && (ppData != NULL);

  if (result)
  {
    *ppData = pThis->pData;
  }
  else if (ppData != NULL)
  {
    *ppData = NULL;
  }
  return result;
}

bool BinaryTreeNode_LeftNode(BinaryTreeNode_t* pThis,
                             BinaryTreeNode_t** ppLeftNode)
{
  bool result = (pThis != NULL) && (ppLeftNode != NULL);

  if (result)
  {
    *ppLeftNode = pThis->pLeftNode;
  }
  else if (ppLeftNode != NULL)
  {
    *ppLeftNode = NULL;
  }

  return result;
}

bool BinaryTreeNode_RightNode(BinaryTreeNode_t* pThis,
                              BinaryTreeNode_t** ppRightNode)
{
  bool result = (pThis != NULL) && (ppRightNode != NULL);

  if (result)
  {
    *ppRightNode = pThis->pRightNode;
  }
  else if (ppRightNode != NULL)
  {
    *ppRightNode = NULL;
  }

  return result;
}
```

3.1.1 Name mangling

As C does not have class or namespace concepts, a different mechanism must be used to indicate that a function should be considered an operation that is part of a class. Different classes often have operations that share the same name, especially if they implement the same interface, and it is essential that two operations may exist in the same project without causing name conflicts.

One solution, employed here and which ensures that C function names are unique, is to derive the function name from the class name and the operation name, joined together by an underscore character. The `data()` operation of the `BinaryTreeNode` class thus becomes `BinaryTreeNode_Data()`.

This is a form of manual *name mangling* (also called *name decoration*). Name mangling is often used by compilers targeting native code to encode additional information about a function as part of its runtime name. The information encoded depends on the calling convention used, and may include, for instance, information on the arguments and return value of a function. Name mangling can be used to encode data not recognized by traditional file formats for executable files and the tools that operate on them (such as linkers). The mangling of compile-time names used here ensures that there are no naming conflicts and adds the name of the class as an aid to human developers.

3.1.2 Error handling

Exceptions change the normal flow of execution when abnormal events occur. Due to the lack of language-level support for exceptions in C, a different means of expressing that an operation could not successfully complete a request is needed. The standard library functions `setjmp()` and `longjmp()` are sometimes used to replicate the exception mechanism of other languages, but this approach is cumbersome to use and has been found to have poor performance (Jung et al. 2008).[3] A simpler approach is to use the return value of operations for error information, and use an output argument for the true return value of the operation (if any). This technique does have the unfortunate side effect of requiring that return values are checked after each and every invocation, as seen in Listing 3.3.

Error information can be provided in a variety of ways:

- **Boolean return value.** The simplest approach is arguably to require that operations return a boolean value indicating whether the invocation was successful. With this approach, it is not possible to easily determine the source of an error, to differentiate between different errors or to present human-readable information about the error.

- **Enumerator return value.** Using an enumerated type makes it possible to differentiate between different errors, while retaining most of the simplicity of using a boolean return value.

- **Integer return value.** An integer can obviously hold the same information as a boolean value or an enumerator. Additional information may be conveyed by dividing the integer space into different regions—for instance, the most significant bit can be used to signal success or failure, the next four bits can be used to communicate the severity of the error, and the remaining bits used to index into a table containing static information on various errors, including localized error messages.

- **Object return value.** The most flexible way of conveying information on an error, of the four options presented here, is to return a reference to an object if an error occurred, or `NULL` otherwise. Such an object can contain a wealth of information on the error, possibly including a stack trace and localized error messages. The downside to this flexibility is that memory must likely be allocated dynamically, which the client must take care to deallocate.

Despite the shortcomings of using a boolean return value, this is the approach used here, due to its simplicity.

[3] Specifically, elaborate runtime book-keeping is needed to ensure that nested "catch" clauses work as intended, and clean-up, in the form of "finally" sections, is cumbersome to implement. In addition, this approach does not work well with component technology, as an exception may be thrown by an operation written in one language, and caught by an operation written in another, possibly interpreted, language.

3.1.3 Instance data

An object combines state and behavior. The state of an object, often referred to as its instance data, is a collection of data operated on by imperative code (the behavior). The most natural way to represent instance data in C is as a structure, and this is the approach taken here. When an object is instantiated, memory is allocated for the instance data on the heap. (Many languages with built-in object models, such as C++, also allow objects to be allocated statically on the call stack. Stack-allocated objects are not considered in this thesis.)

`BinaryTreeNode_Create()` and `BinaryTreeNode_Destroy()` in Listing 3.2 and Listing 3.3 serve as constructor and destructor, respectively, hiding the means used to allocate and deallocate space for the instance data of objects. In this example, all instance variables of the `BinaryTreeNode` class are accessible only to the class implementation (as indicated by their `private` Java access specifiers). As a result, the members of the structure are hidden in the implementation file, conceptually inaccessible from other code.

Operations operate on the instance data, and thus need a way to reference it. A reference to the instance data is given as the first argument, which is normally hidden in languages that natively support object-orientation, such as C++. (Component Pascal is one of the few such languages that makes no attempt to hide this argument (Szyperski et al. 2002:331).) This argument is known as *this* or *self*; in Listing 3.2 and Listing 3.3, this argument is named `pThis`.

3.2 A syntax tree representing an arithmetic expression in C

Arithmetic expressions are traditionally written using infix notation, in which operators are written between operands, and parentheses and precedence rules are used to resolve ambiguity (*5 + 2/3* is a simple example). As most popular programming languages use infix notation, compiler front-ends must parse such expressions (typically using so-called operator-precedence parsers). The result of such a process is often an unambiguous syntax tree that can be used for further processing. The nodes of a syntax tree for arithmetic expressions are the focus of this example.

Four different classes of nodes are used here:

- **Binary operator nodes.** Such nodes hold two operands and a binary operator, such as addition, subtraction, multiplication or division. The operands may be arbitrary nodes. A binary operator node is considered to be constant if both its operands are constant.

- **Unary operator nodes.** Such nodes hold only one operand and a unary operator, such as negation or a trigonometric operator. The operand may be an arbitrary node. A unary operator is considered to be constant if its sole operand is constant.

- **Literal operand nodes.** Such nodes hold a literal, constant value. A literal operand node is considered to be constant at all times, per definition.

- **Identifier operand nodes.** Such nodes hold the name of an identifier, which may be either a variable or a constant. An identifier operand node is considered to be constant if, and only if, the identifier it refers to is a constant.

Figure 3.1 shows one possible syntax tree that may result after parsing a sample arithmetic expression. Constant nodes are shown in gray. (Being able to differentiate between constant

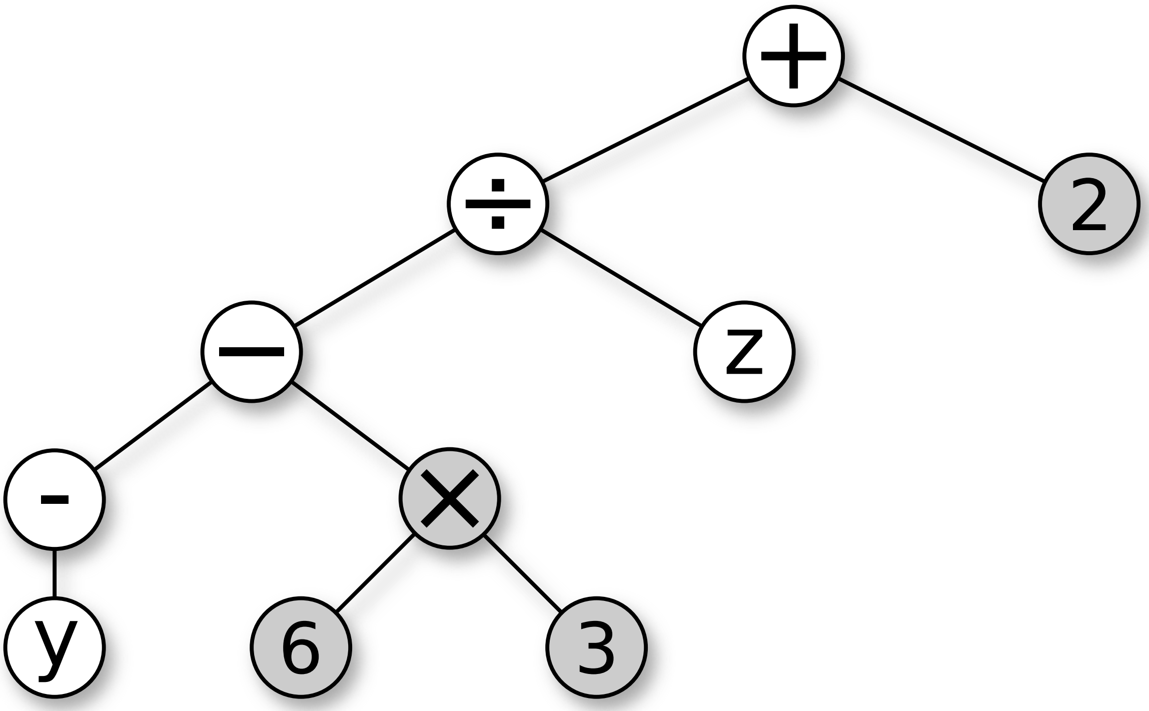

Figure 3.1 Syntax tree corresponding to the expression *((-y - 6 * 3) / z) + 2*

and variable nodes allows for a common optimization known as *constant folding*, which entails replacing fully constant subtrees with literal operand nodes representing their values. The expression of Figure 3.1 would be written *((-y - 18) / z) + 2* after folding constants.)[4]

Nodes are naturally modeled as objects, instantiated from one of four classes, which in turn serve as the default implementations of four interfaces, corresponding to the classes of nodes listed on the facing page. This is depicted in Figure 3.2, using Unified Modeling Language (UML) syntax. For instance, objects modeling binary operator nodes are instances of classes implementing the `BinaryOperatorNode` interface. All objects modeling nodes also implement the `Node` interface, as all four domain-specific node interfaces extend this interface. All nodes may be explicitly destroyed, and all nodes may be queried as to whether they and their children are fully constant. The `DefaultBinaryOperatorNode` class is the sole implementation of the `BinaryOperatorNode` interface.

This example is interesting in that it is highly dependent on dynamic dispatch. An instance of `DefaultBinaryOperatorNode` holds two operands, of which the only known fact is that

[4] This simplistic approach to constant folding will not necessarily work for all arithmetic expressions. The expression *x + 4 + 3* may be correctly represented as a binary operator node adding *3* to another binary operator node, which adds *x* to *4*, in other words *(x + 4) + 3*. Neither binary operator node is considered constant in such a syntax tree, and thus no folding of constants will take place. This problem may be solved by rearranging nodes in such a way that constant nodes (such as *4* and *3*) are grouped together, while respecting the rules governing the operators. As addition is an associative operator, *(x + 4) + 3* can also be written as *x + (4 + 3)*, which may be represented as a binary operator node adding *x* to another binary operator node, which adds *4* to *3*. The latter node is constant, and may be replaced with the literal operand node *7*, yielding the expression *x + 7*.

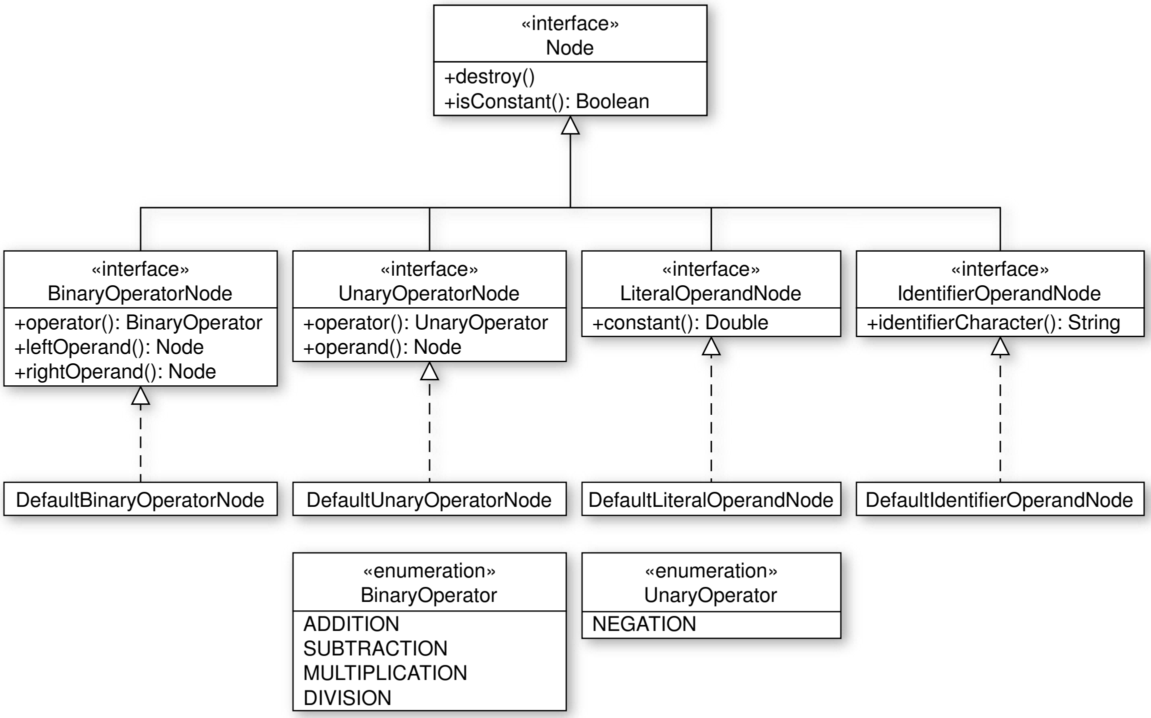

Figure 3.2 Diagram of an interface hierarchy for arithmetic nodes

they implement the `Node` interface. When `Node::IsConstant()` is called on an instance of `DefaultBinaryOperatorNode`, it recursively calls this operation on its two operands, and returns `true` if, and only if, both return `true`. It is unaware of the precise implementation used by its operands, and must therefore find the implementation at runtime, bringing late binding into play.

3.2.1 Introducing late binding

Late binding implies that the implementation of a type must be found at runtime. There are a variety of means to achieve this goal—common to them all is that a data structure needs to be queried at runtime to locate the desired implementation. A reference to this data structure is stored as a (usually) hidden part of the instance data of an object, and is used behind the scenes to locate the implementation.

Listing 3.4 presents a simple Perl script that demonstrates late binding (without object semantics). The script accepts a command-line argument that must be either of the strings "first" or "second," and picks an implementation of the "Foo" operation based on this argument. It achieves this by using a data structure in the form of a Perl associative array (a hash), which maps keys to values, in this case strings to subroutines. As the data structure is used to dispatch subroutine calls, it is called a *dispatch table.*

It would be possible, but inadvisable, to use the same solution in C—that is, a hash table mapping operation names to function addresses. Using the names of operations—strings—is inefficient and often not possible for native code (as the names are typically not available

Listing 3.4 dispatch.pl

```perl
use strict;

my %first_dispatch_table = (
  foo_sub => sub { print "Foo operation, first dispatch table.\n" },
  bar_sub => sub { print "Bar operation, first dispatch table.\n" }
);

my %second_dispatch_table = (
  foo_sub => sub { print "Foo operation, second dispatch table.\n" },
  bar_sub => sub { print "Bar operation, second dispatch table.\n" }
);

# Retrieve the desired implementation subroutine.
my $implementation_sub;

if (@ARGV == 1) {
  if ($ARGV[0] eq "first") {
    $implementation_sub = $first_dispatch_table{foo_sub};
  } elsif ($ARGV[0] eq "second") {
    $implementation_sub = $second_dispatch_table{foo_sub};
  }
}

die "Cannot find implementation.\n" if !defined($implementation_sub);

# Dispatch the call.
$implementation_sub->();
```

at runtime). The standard solution is to use tables of procedural variables—structures of function pointers, in C parlance—indexed into by indices that are known at compile-time. Calling an operation using such a table is obviously not as efficient as eliminating dynamic dispatch entirely *(static dispatch)*, but more efficient than using more elaborate data structures, such as the hash table used in Listing 3.4. This solution is very common, and is known by many names, including *virtual method table (VMT)*, *virtual table (VT, VTBL)*, *vtable*, *virtual function table* and dispatch table (the plethora of names containing the word "virtual" is due to the use of dispatch tables to implement virtual operations in languages such as C++).

Listing 3.5 shows the C header file for the `Node` interface. The specification for the `Node` dispatch table appears at the bottom as the `Node_DispatchTable_t` type, and declares two function pointers consistent with Figure 3.2.[5] The `Node_t` type stipulates that for a pointer to be considered a reference to an object implementing the `Node` interface, the first member of the memory area it points to must be a pointer to a dispatch table compatible with `Node_DispatchTable_t`. The memory layout stipulated by these two types is an example of

[5] Calls to `Node::Destroy()`, and other destructors in this thesis, are guaranteed to succeed at all times. Destructors that may fail are difficult to handle well, especially when called from other destructors (if a destructor destroys some of its members before encountering a member destructor that fails, it is difficult to recover).

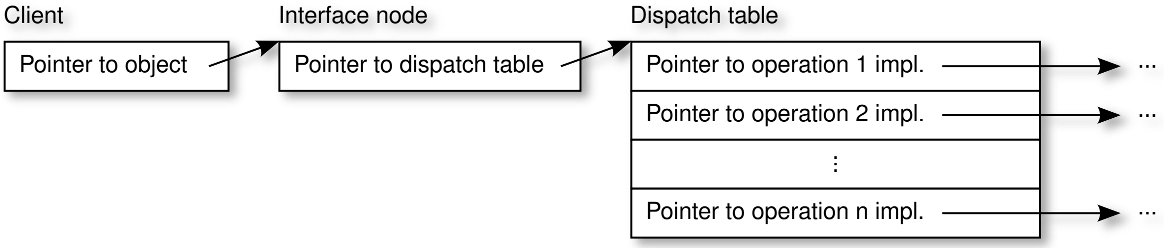
Client
Pointer to object
Interface node
Pointer to dispatch table
Dispatch table
Pointer to operation 1 impl.
Pointer to operation 2 impl.
⋮
Pointer to operation n impl.
···

Figure 3.3 Memory layout of interfaces of the custom object model

one part of the binary standard for the object model designed in this chapter.[6] It is depicted in Figure 3.3. The convenience macros `Node_Destroy()` and `Node_IsConstant()` help clients make calls to objects implementing this interface.

Listing 3.5 Node.h

```
/**
 * @file
 *
 * This file contains an interface, <code>Node</code>, representing an
 * immutable node in a syntax tree. This tree represents an arithmetic
 * expression as produced by an operator-precedence parser. This
 * interface is not meant to be implemented directly; rather, it
 * stipulates what functionality all nodes in said tree are expected
 * to provide. Descendant interfaces add operations specific to the
 * type of node they are modeling.
 */

#ifndef INCLUSION_GUARD_NODE
#define INCLUSION_GUARD_NODE

#include <stdbool.h>

// Convenience macros for the operations defined in this interface:

/**
 * Destroys this node and its children (if any). This operation always
 * succeeds.
 *
 * @param[in] pNode
 *   the instance implementing this interface. May be
 *   <code>NULL</code>.
 */
#define Node_Destroy(pNode)                                              \
  if ((pNode) != NULL)                                                   \
  {                                                                      \
    (pNode)->pDispatchTable->Destroy(pNode);                             \
  }
```

[6] Having knowledge of the memory layout is necessary, but not sufficient, to make use of objects that conform to this object model. See section 4.4.

```
/**
 * Returns <code>true</code> if this node is fully constant and
 * <code>false</code> otherwise.
 *
 * @param[in] pNode
 *   the instance implementing this interface. Must not be
 *   <code>NULL</code>.
 * @param[out] pResult
 *   a pointer to the variable which shall hold the result returned by
 *   this operation, that is, whether this node is constant. Must not
 *   be <code>NULL</code>.
 * @return
 *   <code>true</code> if the operation completes successfully,
 *   <code>false</code> otherwise.
 */
#define Node_IsConstant(pNode, pResult)                                \
  (pNode)->pDispatchTable->IsConstant((pNode), (pResult))

struct Node_DispatchTable_s;

/**
 * Variables of this type may be used to represent objects
 * implementing this interface.
 */
typedef struct
{
  const struct Node_DispatchTable_s* pDispatchTable;
} Node_t;

/**
 * This is the dispatch table, or vtable, for this interface. It
 * represents an indirection that enables the implementation of an
 * interface to be bound to at runtime. Interfaces wishing to extend
 * this interface must include the complete contents of this structure
 * as the first members of their dispatch tables, changing the
 * <code>Node_t</code> type to match their own. (Only single interface
 * inheritance is supported.)
 */
typedef struct Node_DispatchTable_s
{
  // Operations defined in the Node interface:
  void (*Destroy)(Node_t* pNode);
  bool (*IsConstant)(Node_t* pNode, bool* pResult);
} Node_DispatchTable_t;

#endif // INCLUSION_GUARD_NODE
```

Dispatch tables compatible with the `Node` interface must not necessarily be of the formal compile-time type `Node_DispatchTable_t`, and a proper `Node` object reference must not necessarily be of the type `Node_t`. They must, however, be binary compatible with these types; in other words, regardless of the formal compile-time types of `Node` references and `Node` dispatch tables, it must be possible to treat them as though they were of the types described here. The upshot is that a dispatch table compatible with the `Node_DispatchTable_t` type may include data following that specified by the aforementioned type, and ditto for object references compatible with the `Node_t` type.

This aspect makes it straightforward for another interface to inherit from `Node`—its dispatch table type must simply include all members of the `Node_DispatchTable_t` type before its own members, changing the `Node_t` type to match its own type (which is binary compatible with `Node_t`).[7] Listing 3.6 shows the `BinaryOperatorNode` interface, which extends `Node`. (It is dependent on the enumerated type `BinaryOperator_t`, whose members are shown in Figure 3.2 on page 34.) The `BinaryOperatorNode_DispatchTable_t` and `BinaryOperatorNode_t` types are fully binary compatible with the `Node_DispatchTable_t` and `Node_t` types defined in Listing 3.5.

Listing 3.6 BinaryOperatorNode.h

```
/**
 * @file
 *
 * This file contains an interface, <code>BinaryOperatorNode</code>,
 * representing a binary operator node in a syntax tree. This tree
 * represents an arithmetic expression. A binary operator takes two
 * operands (in the expression "3 / x", the division operator operates
 * on the literal operand "3" and the identifier operand "x"). This
 * interface extends <code>Node</code>.
 */

#ifndef INCLUSION_GUARD_BINARY_OPERATOR_NODE
#define INCLUSION_GUARD_BINARY_OPERATOR_NODE

#include <stdbool.h>
#include "Node.h"
#include "BinaryOperator.h"

/* Convenience macros for the operations inherited from the Node
 * interface:
 */
#define BinaryOperatorNode_Destroy(pBinaryOperatorNode)                 \
   if ((pBinaryOperatorNode) != NULL)                                   \
   {                                                                    \
      (pBinaryOperatorNode)->pDispatchTable->Destroy(pBinaryOperatorNode);\
   }
#define BinaryOperatorNode_IsConstant(pBinaryOperatorNode, pResult)     \
   (pBinaryOperatorNode)->pDispatchTable->IsConstant(                   \
      (pBinaryOperatorNode), (pResult))

// Convenience macros for the operations defined in this interface:

/**
 * Returns the binary operator of this node.
 *
```

[7] This approach only works when extending a single interface, known as *single interface inheritance*. Extending multiple interfaces is discussed in Chapter 4.

```
 * @param[in] pBinaryOperatorNode
 *   the instance implementing this interface. Must not be
 *   <code>NULL</code>.
 * @param[out] pResult
 *   a pointer to the variable which shall hold the result returned by
 *   this operation, that is, the binary operator of this node.
 * @return
 *   <code>true</code> if the operation completes successfully,
 *   <code>false</code> otherwise.
 */
#define BinaryOperatorNode_Operator(pBinaryOperatorNode, pResult)       \
  (pBinaryOperatorNode)->pDispatchTable->Operator((pBinaryOperatorNode),\
                                                  (pResult))

/**
 * Returns the left operand of this node.
 *
 * @param[in] pBinaryOperatorNode
 *   the instance implementing this interface. Must not be
 *   <code>NULL</code>.
 * @param[out] ppResult
 *   a pointer to the variable which shall hold the result returned by
 *   this operation, that is, the left operand of this node.
 * @return
 *   <code>true</code> if the operation completes successfully,
 *   <code>false</code> otherwise.
 */
#define BinaryOperatorNode_LeftOperand(pBinaryOperatorNode, ppResult)   \
  (pBinaryOperatorNode)->pDispatchTable->LeftOperand(                   \
    (pBinaryOperatorNode),                                              \
    (ppResult))

/**
 * Returns the right operand of this node.
 *
 * @param[in] pBinaryOperatorNode
 *   the instance implementing this interface. Must not be
 *   <code>NULL</code>.
 * @param[out] ppResult
 *   a pointer to the variable which shall hold the result returned by
 *   this operation, that is, the right operand of this node.
 * @return
 *   <code>true</code> if the operation completes successfully,
 *   <code>false</code> otherwise.
 */
#define BinaryOperatorNode_RightOperand(pBinaryOperatorNode, ppResult)  \
  (pBinaryOperatorNode)->pDispatchTable->RightOperand(                  \
    (pBinaryOperatorNode),                                              \
    (ppResult))

struct BinaryOperatorNode_DispatchTable_s;
```

```
/**
 * Variables of this type may be used to represent objects
 * implementing this interface.
 */
typedef struct
{
  const struct BinaryOperatorNode_DispatchTable_s* pDispatchTable;
} BinaryOperatorNode_t;

/**
 * This is the dispatch table, or vtable, for this interface. It
 * represents an indirection that enables the implementation of an
 * interface to be bound to at runtime. Interfaces wishing to extend
 * this interface must include the complete contents of this structure
 * as the first members of their dispatch tables, changing the
 * <code>BinaryOperatorNode_t</code> type to match their own. (Only
 * single interface inheritance is supported.)
 */
typedef struct BinaryOperatorNode_DispatchTable_s
{
  // Operations inherited from the Node interface:
  void (*Destroy)(BinaryOperatorNode_t* pBinaryOperatorNode);
  bool (*IsConstant)(BinaryOperatorNode_t* pBinaryOperatorNode,
                     bool* pResult);

  // Operations defined in the BinaryOperatorNode interface:
  bool (*Operator)(BinaryOperatorNode_t* pBinaryOperatorNode,
                   BinaryOperator_t* pResult);
  bool (*LeftOperand)(BinaryOperatorNode_t* pBinaryOperatorNode,
                      Node_t** ppResult);
  bool (*RightOperand)(BinaryOperatorNode_t* pBinaryOperatorNode,
                       Node_t** ppResult);
} BinaryOperatorNode_DispatchTable_t;

#endif // INCLUSION_GUARD_BINARY_OPERATOR_NODE
```

Writing a class that implements the `BinaryOperatorNode` interface (and thus also the `Node` interface) entails complying with the binary standard of this interface, in the form of the `BinaryOperatorNode_DispatchTable_t` and `BinaryOperatorNode_t` types. A pointer to an instance of such a class must be binary compatible with the latter type, and its `pDispatchTable` member must be binary compatible with the former type.

`DefaultBinaryOperatorNode` is such a class. All its operations are virtual, except its constructor (constructors are always bound to statically, as clients are always aware of the identities of the classes they instantiate). Its header file appears as Listing 3.7 and its implementation as Listing 3.8. Figure 3.4 depicts the instance data of `DefaultBinaryOperatorNode` objects, which are fully compatible with the binary standard shown in Figure 3.3 on page 36. The contents of the white areas in the figure are dictated by the binary interface standard—the first member of the memory representing an object must be a pointer to a dispatch table, the layout of which is fixed—whereas the shaded areas may contain any data that is convenient for the implementation. This latter space is used for the instance data.

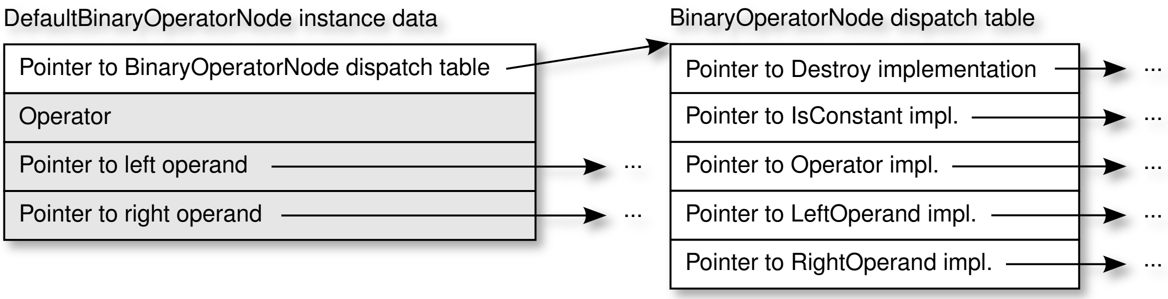
DefaultBinaryOperatorNode instance data
Pointer to BinaryOperatorNode dispatch table
Operator
Pointer to left operand
Pointer to right operand
...
...
BinaryOperatorNode dispatch table
Pointer to Destroy implementation
Pointer to IsConstant impl.
Pointer to Operator impl.
Pointer to LeftOperand impl.
Pointer to RightOperand impl.
...
...
...
...
...

Figure 3.4 Instance data of `DefaultBinaryOperatorNode` objects

Listing 3.7 DefaultBinaryOperatorNode.h

```
/**
 * @file
 *
 * This file contains function prototypes for the class operations
 * (static operations) of the <code>DefaultBinaryOperatorNode</code>
 * class.
 */

#ifndef INCLUSION_GUARD_DEFAULT_BINARY_OPERATOR_NODE
#define INCLUSION_GUARD_DEFAULT_BINARY_OPERATOR_NODE

#include <stdbool.h>
#include "Node.h"
#include "BinaryOperatorNode.h"
#include "BinaryOperator.h"

/**
 * Creates an instance of the <code>DefaultBinaryOperatorNode</code>
 * class.
 *
 * @param[in] operator
 *   the binary operator of this node.
 * @param[in] pLeftOperand
 *   the left operand of this node. Must not be <code>NULL</code>.
 * @param[in] pRightOperand
 *   the right operand of this node. Must not be <code>NULL</code>.
 * @param[out] ppResult
 *   a pointer to the variable which shall hold the instance of this
 *   class. Must not be <code>NULL</code>.
 * @return
 *   <code>true</code> if the operation completes successfully,
 *   <code>false</code> otherwise.
 */
bool DefaultBinaryOperatorNode_Create(BinaryOperator_t operator,
                                      Node_t* pLeftOperand,
                                      Node_t* pRightOperand,
                                      BinaryOperatorNode_t** ppResult);
```

```
#endif // INCLUSION_GUARD_DEFAULT_BINARY_OPERATOR_NODE
```

Listing 3.8 DefaultBinaryOperatorNode.c

```
/**
 * @file
 *
 * This file contains a class, <code>DefaultBinaryOperatorNode</code>,
 * which is the default implementation of the
 * <code>BinaryOperatorNode</code> interface.
 */

#include <stdbool.h>
#include <stdlib.h>
#include <stdarg.h>
#include "Node.h"
#include "BinaryOperatorNode.h"
#include "BinaryOperator.h"
#include "DefaultBinaryOperatorNode.h"

/**
 * This structure represents the instance data of
 * <code>DefaultBinaryOperatorNode</code> objects.
 */
typedef struct
{
  /**
   * The dispatch table used by this instance.
   */
  const BinaryOperatorNode_DispatchTable_t* pDispatchTable;

  /**
   * The binary operator used by this object.
   */
  BinaryOperator_t operator;

  /**
   * The node representing the left operand. This member is never
   * <code>NULL</code>.
   */
  Node_t* pLeftOperand;

  /**
   * The node representing the right operand. This member is never
   * <code>NULL</code>.
   */
  Node_t* pRightOperand;
} DefaultBinaryOperatorNode_InstanceData_t;

static const BinaryOperatorNode_DispatchTable_t
  gBinaryOperatorNodeDispatchTable;
```

```
bool DefaultBinaryOperatorNode_Create(BinaryOperator_t operator,
                                      Node_t* pLeftOperand,
                                      Node_t* pRightOperand,
                                      BinaryOperatorNode_t** ppResult)
{
  DefaultBinaryOperatorNode_InstanceData_t* pInstanceData = NULL;

  bool result =
    (pLeftOperand != NULL) &&
    (pRightOperand != NULL) &&
    (ppResult != NULL);

  if (result)
  {
    pInstanceData =
      malloc(sizeof(DefaultBinaryOperatorNode_InstanceData_t));
    result = pInstanceData != NULL;
  }

  if (result)
  {
    pInstanceData->pDispatchTable = &gBinaryOperatorNodeDispatchTable;
    pInstanceData->operator = operator;
    pInstanceData->pLeftOperand = pLeftOperand;
    pInstanceData->pRightOperand = pRightOperand;
    *ppResult = (BinaryOperatorNode_t*)pInstanceData;
  }
  else if (ppResult != NULL)
  {
    *ppResult = NULL;
  }

  return result;
}

static void DefaultBinaryOperatorNode_Destroy(
  BinaryOperatorNode_t* pBinaryOperatorNode)
{
  if (pBinaryOperatorNode != NULL)
  {
    DefaultBinaryOperatorNode_InstanceData_t* pThis =
      (DefaultBinaryOperatorNode_InstanceData_t*)pBinaryOperatorNode;

    Node_Destroy(pThis->pLeftOperand);
    Node_Destroy(pThis->pRightOperand);

    free(pBinaryOperatorNode);
  }
}

static bool DefaultBinaryOperatorNode_IsConstant(
  BinaryOperatorNode_t* pBinaryOperatorNode, bool* pResult)
{
  bool result = (pBinaryOperatorNode != NULL) && (pResult != NULL);
  DefaultBinaryOperatorNode_InstanceData_t* pThis =
    (DefaultBinaryOperatorNode_InstanceData_t*)pBinaryOperatorNode;
  bool isLeftOperandConstant = false;
  bool isRightOperandConstant = false;

  if (result)
  {
    result = Node_IsConstant(pThis->pLeftOperand,
                             &isLeftOperandConstant);
  }

  if (result)
  {
    result = Node_IsConstant(pThis->pRightOperand,
                             &isRightOperandConstant);
  }

  if (result)
  {
    *pResult = isLeftOperandConstant && isRightOperandConstant;
  }
  else if (pResult != NULL)
  {
    *pResult = false;
  }

  return result;
}

static bool DefaultBinaryOperatorNode_Operator(
  BinaryOperatorNode_t* pBinaryOperatorNode,
  BinaryOperator_t* pResult)
{
  bool result = (pBinaryOperatorNode != NULL) && (pResult != NULL);
  DefaultBinaryOperatorNode_InstanceData_t* pThis =
    (DefaultBinaryOperatorNode_InstanceData_t*)pBinaryOperatorNode;

  if (result)
  {
    *pResult = pThis->operator;
  }
  else if (pResult != NULL)
  {
    *pResult = BinaryOperator_UNDEFINED;
  }

  return result;
}

static bool DefaultBinaryOperatorNode_LeftOperand(
  BinaryOperatorNode_t* pBinaryOperatorNode,
  Node_t** ppResult)
```

```
{
  bool result = (pBinaryOperatorNode != NULL) && (ppResult != NULL);
  DefaultBinaryOperatorNode_InstanceData_t* pThis =
    (DefaultBinaryOperatorNode_InstanceData_t*)pBinaryOperatorNode;

  if (result)
  {
    *ppResult = pThis->pLeftOperand;
  }
  else if (ppResult != NULL)
  {
    *ppResult = NULL;
  }

  return result;
}

static bool DefaultBinaryOperatorNode_RightOperand(
  BinaryOperatorNode_t* pBinaryOperatorNode,
  Node_t** ppResult)
{
  bool result = (pBinaryOperatorNode != NULL) && (ppResult != NULL);
  DefaultBinaryOperatorNode_InstanceData_t* pThis =
    (DefaultBinaryOperatorNode_InstanceData_t*)pBinaryOperatorNode;

  if (result)
  {
    *ppResult = pThis->pRightOperand;
  }
  else if (ppResult != NULL)
  {
    *ppResult = NULL;
  }

  return result;
}

/**
 * This is the dispatch table for the <code>BinaryOperatorNode</code>
 * interface implemented by this class.
 */
static const BinaryOperatorNode_DispatchTable_t
  gBinaryOperatorNodeDispatchTable =
{
  DefaultBinaryOperatorNode_Destroy,
  DefaultBinaryOperatorNode_IsConstant,
  DefaultBinaryOperatorNode_Operator,
  DefaultBinaryOperatorNode_LeftOperand,
  DefaultBinaryOperatorNode_RightOperand
};
```

The `DefaultBinaryOperatorNode_InstanceData_t` type on page 42 represents the instance data of `DefaultBinaryOperatorNode` objects. The first member is a pointer to the dispatch table, in accordance with the `BinaryOperatorNode_t` type on page 40. The members declared after the pointer to the dispatch table are used for the actual instance data.

All instances of a certain class use the same dispatch table, as the only thing that varies between objects of the same class is their state. The behavior of different objects of the same class differs only insofar as they have different state, as they use the same implementation. As such, the dispatch table is statically allocated as the constant structure `gBinaryOperatorNodeDispatchTable` at the end of Listing 3.8 on the previous page. All instances of a class point to the same global dispatch table. While the formal compile-time type of this variable is `BinaryOperatorNode_DispatchTable_t`, it is binary compatible with `Node_DispatchTable_t` as well.

Except for the constructor, all functions are declared `static`, and as a result their symbol names are not available outside the compilation unit in which they are defined. They are, however, still accessible to other compilation units as their addresses are stored in the dispatch table. Essentially, this approach subverts the standard means that C provides for calling external functions (external linkage), enabling the implementation of dispatch in an object-oriented fashion.

All operations accept a pointer to a variable of the `BinaryOperatorNode_t` type. This variable needs to be typecast to the `DefaultBinaryOperatorNode_InstanceData_t` type in order to access the instance data (predictably, the local variable used to access the instance data is called `pThis`).

This example program essentially implements a complete, if simple, object model through the conventions it adopts. There is room for improvement, though, which is the topic for Chapter 4.

CHAPTER 4

Refining the object model

The object model presented in Chapter 3 successfully realizes classes, interfaces, encapsulation, polymorphism and single interface inheritance. However, classes may only directly implement one single interface. This is limiting from a modeling perspective—an object should be able to signal that it is both "observable" and "printable," for instance. If a class is allowed to directly implement multiple interfaces, though, there must be a way for clients to query an object as to whether it supports a given interface, and if so, for a reference through this interface (known as *interface navigation*).

The `Node` interface in Chapter 3 stipulates that explicit memory management should be used to manage the lifetime of instances implementing this interface and its descendants. This strategy is inadequate in a setting involving many independent parties. If a reference to an object implementing the `Node` interface is passed to an external party, there is no way of knowing if this party wishes to hold onto the reference past the lifetime of the call (it could store it as part of its instance data). As a result, it is impossible to know when it is safe to destroy the object. Reference counting can be used to solve this problem, as described in section 2.2.5.[1]

This chapter aims to rectify these deficiencies, by introducing support for interface navigation and reference counting to the `DefaultBinaryOperatorNode` class presented in Chapter 3. To this end, a root interface, `Fundamental`, is introduced in this chapter that all interfaces must extend.

Using the `DefaultBinaryOperatorNode` class through the `BinaryOperatorNode` interface requires compile-time knowledge of this interface. The validity of such invocations are checked against the function prototypes contained in the dispatch table of `BinaryOperatorNode`. As noted in section 2.2.4, not all clients that wish to communicate with an object necessarily have compile-time knowledge of the interfaces it implements, notably scripts interpreted or compiled at runtime, which may not even have been written at the time the script host was compiled. Very late binding entails making it possible to check the validity of operation invocations at runtime, by making the metadata previously only available at compile-time also available at runtime, and thus deploying it to end-users' systems (this data is sometimes known as runtime type information). This chapter adds support for very late binding to the `DefaultBinaryOperatorNode` class, by having it implement a new interface, `Scriptable`, which provides an operation that facilitates very late binding.

[1] The problem with cyclic references would be provoked by adding a parent node reference to every node.

4.1 Instituting a root interface

The new root interface, `Fundamental`, features three operations that facilitate interface navigation and reference counting. As all interfaces must directly or indirectly descend from this interface, all objects gain these new abilities. The revised interface hierarchy is shown in Figure 4.1. (The `Destroy()` operation has been removed from the `Node` interface in favor of the reference counting operations inherited from the root interface, and a new operation, `PrintDebugInformation()`, has been added to aid debugging.)

Listing 4.1 presents the C header file for the new base interface, along with documentation for the new mandatory operations. Figure 4.2 visually depicts the revised memory layout and its requirement that the first three members of all dispatch tables correspond to the three operations of the `Fundamental` interface.

Listing 4.1 Fundamental.h

```
/**
 * @file
 *
 * This file contains the interface <code>Fundamental</code>, which
 * all interfaces ultimately descend from. It contains operations that
 * facilitate interface navigation and reference counting.
 */

#ifndef INCLUSION_GUARD_FUNDAMENTAL
#define INCLUSION_GUARD_FUNDAMENTAL

#include <stdbool.h>
#include <wchar.h>

/**
 * The name of this interface.
 */
#define FUNDAMENTAL_NAME (L"se.polberger.components.Fundamental")

/**
 * Returns a reference to the underlying instance through a different
 * interface. <code>::AddReference()</code> is called automatically on
 * the new reference. If the class does not support the given
 * interface, this operation fails.
 *
 * @param[in] pFundamental
 *    the instance implementing this interface. Must not be
 *    <code>NULL</code>.
 * @param[in] pInterfaceName
 *    the name of the interface. If this interface is not implemented
 *    by the underlying instance, this operation fails. Must not be
 *    <code>NULL</code>.
 * @param[out] ppResult
 *    a pointer to the variable which shall hold the result returned by
 *    this operation, that is, an interface reference. This parameter
 *    may be <code>NULL</code>, in which case this operation may be
 *    used to query an object as to whether it supports a given
 *    interface, with no concern for a result interface.
```

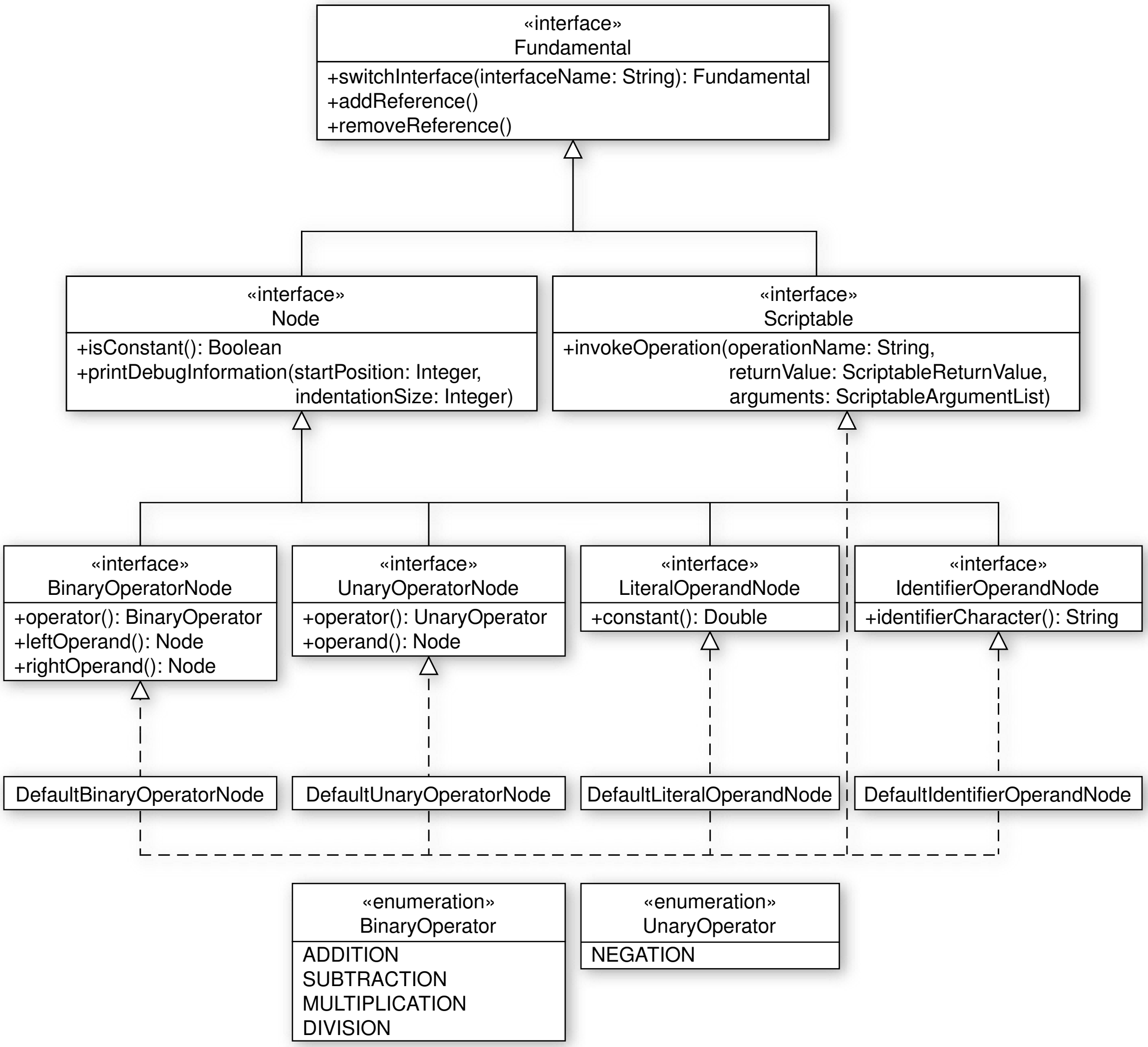

Figure 4.1 Revised diagram of an interface hierarchy for arithmetic nodes

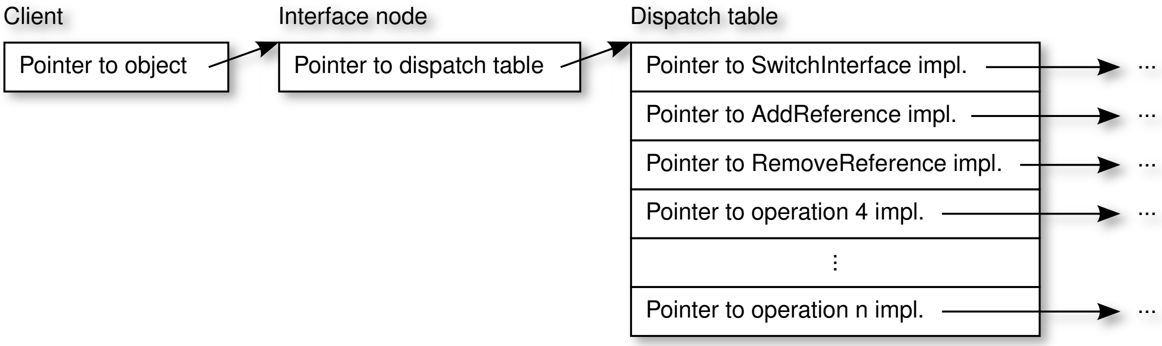
Client
Pointer to object
Interface node
Pointer to dispatch table
Dispatch table
Pointer to SwitchInterface impl.
Pointer to AddReference impl.
Pointer to RemoveReference impl.
Pointer to operation 4 impl.
⋮
Pointer to operation n impl.
...
...
...
...
...

Figure 4.2 Revised memory layout of interfaces of the custom object model

```
 * @return
 *   <code>true</code> if the operation completes successfully,
 *   <code>false</code> otherwise.
 */
#define Fundamental_SwitchInterface(pFundamental,                      \
                                    pInterfaceName,                    \
                                    ppResult)                          \
  (pFundamental)->pDispatchTable->SwitchInterface((pFundamental),      \
                                                  (pInterfaceName),    \
                                                  (ppResult));

/**
 * Notifies the instance implementing this interface that a new
 * reference has been added. This call always succeeds (thus there is
 * no return value).
 *
 * @param[in] pFundamental
 *   the instance implementing this interface. May be
 *   <code>NULL</code>.
 */
#define Fundamental_AddReference(pFundamental)                         \
  if ((pFundamental) != NULL)                                          \
  {                                                                    \
    (pFundamental)->pDispatchTable->AddReference(pFundamental);        \
  }

/**
 * Notifies the instance implementing this interface that a previously
 * added reference has been removed. If this instance finds that there
 * are no live references to it, it automatically destroys
 * itself. This call always succeeds (thus there is no return value).
 *
 * @param[in] pFundamental
 *   the instance implementing this interface. May be
 *   <code>NULL</code>.
 */
#define Fundamental_RemoveReference(pFundamental)                      \
  if ((pFundamental) != NULL)                                          \
  {                                                                    \
    (pFundamental)->pDispatchTable->RemoveReference(pFundamental);     \
  }

struct Fundamental_DispatchTable_s;

/**
 * Variables of this type may be used to represent objects
 * implementing this interface.
 */
typedef struct
{
  const struct Fundamental_DispatchTable_s* pDispatchTable;
} Fundamental_t;
```

```
/**
 * This is the dispatch table, or vtable, for this interface. It
 * represents an indirection that enables the implementation of an
 * interface to be bound to at runtime. Interfaces wishing to extend
 * this interface must include the complete contents of this structure
 * as the first members of their dispatch tables, changing the
 * <code>Fundamental_t</code> type to match their own. (Only single
 * interface inheritance is supported.)
 */
typedef struct Fundamental_DispatchTable_s
{
  bool (*SwitchInterface)(Fundamental_t* pFundamental,
                          wchar_t* pInterfaceName,
                          Fundamental_t** ppResult);
  void (*AddReference)(Fundamental_t* pFundamental);
  void (*RemoveReference)(Fundamental_t* pFundamental);
} Fundamental_DispatchTable_t;

#endif // INCLUSION_GUARD_FUNDAMENTAL
```

`Fundamental::SwitchInterface()` enables clients that hold a reference to an object through a certain interface to switch to another interface that the object implements (or, alternatively, to simply query an object as to whether it supports a given interface). To facilitate this, it must be possible to refer to an interface at runtime. In other words, every interface must have a runtime-accessible name. Elaborate means have been devised for component models in industry to give interfaces and other such entities runtime names. For simplicity, this example uses a simple text string, which is understood to have been crafted in such a way that the name has a high probability of being unique. All interfaces used here use the simple, but very effective, method of using Internet top-level domain names as part of the name, thus delegating the responsibility for ensuring the uniqueness of the names to third-party naming authorities (domain name registrars). This is also the approach taken by the Java programming language and platform. The name of the `Fundamental` interface is given as the contents of the `FUNDAMENTAL_NAME` macro on page 48.

`Fundamental::AddReference()` and `Fundamental::RemoveReference()`, which realize the reference counting scheme used by this object model, are called when a reference is added to an object, and when a reference is removed, respectively. A newly-created object assumes that at least one client holds a reference to it.

Using the reference counting operations properly can be error-prone. An object may be passed as an input argument to another operation without adding a reference, but if one is returned to a caller using an output argument, one must be added. Also, if an object reference is saved to a part of the system memory that is outside the call stack (a global variable or the heap), a reference also needs to be added. If these rules are not obeyed, objects may be unexpectedly destroyed, or may continue to consume memory despite no longer being needed.

A class implementing multiple interfaces may want to either provide one constructor per supported interface, or one universal constructor that accepts the name of the desired interface as an argument. If the latter path is taken, the constructor should return a `Fundamental` instance that the client is expected to typecast to the proper type. `DefaultBinaryOperatorNode` uses this approach; the new function prototype and its documentation can be seen in Listing 4.2.

Listing 4.2 Excerpt from a revised version of DefaultBinaryOperatorNode.h

```
/**
 * Creates an instance of the <code>DefaultBinaryOperatorNode</code>
 * class.
 *
 * @param[in] operator
 *   the binary operator of this node.
 * @param[in] pLeftOperand
 *   the left operand of this node. Must not be <code>NULL</code>.
 * @param[in] pRightOperand
 *   the right operand of this node. Must not be <code>NULL</code>.
 * @param[in] pInterfaceName
 *   the name of the interface that the reference should be returned
 *   through. Must not be <code>NULL</code>.
 * @param[out] ppResult
 *   a pointer to the variable which shall hold the instance of this
 *   class. Must not be <code>NULL</code>.
 * @return
 *  <code>true</code> if the operation completes successfully,
 *  <code>false</code> otherwise.
 */
bool DefaultBinaryOperatorNode_Create(BinaryOperator_t operator,
                                      Node_t* pLeftOperand,
                                      Node_t* pRightOperand,
                                      wchar_t* pInterfaceName,
                                      Fundamental_t** ppResult);
```

Apart from requiring all interfaces to directly or indirectly descend from `Fundamental`, the nature of the binary standard has not changed since Chapter 3 to support classes implementing multiple unrelated interfaces. This can be seen in Listing 4.1: The `Fundamental_t` and `Fundamental_DispatchTable_t` types follow the conventions set forth in Chapter 3. Thus, the onus is fully on the implementation of the classes themselves to realize this support.

An excerpt from the revised implementation of `DefaultBinaryOperatorNode` is shown in Listing 4.3. To illustrate the new support for implementing multiple interfaces, this class now also implements the `Scriptable` interface, which descends directly from `Fundamental`. (`Scriptable` makes it possible to call an object without having compile-time knowledge of the interfaces it implements, and is described more fully in section 4.2.)

Listing 4.3 Excerpt 1 from a revised version of DefaultBinaryOperatorNode.c

```
struct DefaultBinaryOperatorNode_InstanceData_s;

/**
 * This structure represents an interface node, which enables access
 * to instances of the class contained in this file.
 */
typedef struct
{
  /**
   * The dispatch table, or vtable. The type is not specified, as this
   * structure is not used to access the dispatch table (whose type
   * may vary), but to access the instance data.
   */
  const void* pDispatchTable;

  /**
   * The instance data.
   */
  struct DefaultBinaryOperatorNode_InstanceData_s* pInstanceData;
} DefaultBinaryOperatorNode_InterfaceNode_t;

/**
 * This structure represents the instance data of
 * <code>DefaultBinaryOperatorNode</code> objects.
 */
typedef struct DefaultBinaryOperatorNode_InstanceData_s
{
  /**
   * The interface node that makes it possible to access this instance
   * through the <code>BinaryOperatorNode</code> interface (and all
   * interfaces that this interface descends from).
   */
  DefaultBinaryOperatorNode_InterfaceNode_t
    interfaceNodeBinaryOperatorNode;

  /**
   * The interface node that makes it possible to access this instance
   * through the <code>Scriptable</code> interface (and all interfaces
   * that this interface descends from).
   */
  DefaultBinaryOperatorNode_InterfaceNode_t interfaceNodeScriptable;

  /**
   * The reference count of this instance. When this drops to zero,
   * the instance is automatically destroyed.
   */
  unsigned int referenceCount;

  /**
   * The binary operator used by this object.
   */
  BinaryOperator_t operator;

  /**
   * The node representing the left operand. This member is never
   * <code>NULL</code>.
   */
  Node_t* pLeftOperand;

  /**
   * The node representing the right operand. This member is never
   * <code>NULL</code>.
   */
  Node_t* pRightOperand;
} DefaultBinaryOperatorNode_InstanceData_t;

static const BinaryOperatorNode_DispatchTable_t
  gBinaryOperatorNodeDispatchTable;
static const Scriptable_DispatchTable_t gScriptableDispatchTable;

bool DefaultBinaryOperatorNode_Create(BinaryOperator_t operator,
                                      Node_t* pLeftOperand,
                                      Node_t* pRightOperand,
                                      wchar_t* pInterfaceName,
                                      Fundamental_t** ppResult)
{
  DefaultBinaryOperatorNode_InstanceData_t* pInstanceData = NULL;

  bool result =
    (pLeftOperand != NULL) &&
    (pRightOperand != NULL) &&
    (pInterfaceName != NULL) &&
    (ppResult != NULL);

  if (result)
  {
    pInstanceData =
      malloc(sizeof(DefaultBinaryOperatorNode_InstanceData_t));
    result = pInstanceData != NULL;
  }

  if (result)
  {
    pInstanceData->interfaceNodeBinaryOperatorNode.pDispatchTable =
      &gBinaryOperatorNodeDispatchTable;
    pInstanceData->interfaceNodeBinaryOperatorNode.pInstanceData =
      pInstanceData;

    pInstanceData->interfaceNodeScriptable.pDispatchTable =
      &gScriptableDispatchTable;
    pInstanceData->interfaceNodeScriptable.pInstanceData =
      pInstanceData;

    pInstanceData->referenceCount = 1;
    pInstanceData->operator = operator;

    /* We are about to make copies of the operand nodes passed to this
     * constructor that we will keep as part of our instance data. As
     * a result, we need to add references to them.
     */
    Node_AddReference(pLeftOperand);
    pInstanceData->pLeftOperand = pLeftOperand;
    Node_AddReference(pRightOperand);
    pInstanceData->pRightOperand = pRightOperand;

    /* Switch to the desired interface. This operation automatically
```

```
     * adds a reference to the reference it returns.
     */
    result = Fundamental_SwitchInterface(
      (Fundamental_t*)&(pInstanceData->interfaceNodeBinaryOperatorNode),
      pInterfaceName,
      ppResult);
  }
  else if (ppResult != NULL)
  {
    *ppResult = NULL;
  }

  /* Regardless of whether we have failed or succeeded in switching to
   * the desired interface, we need to remove the reference this
   * constructor has added to the instance (implicitly, by setting the
   * reference count to 1).
   */
  if (pInstanceData != NULL)
  {
    Fundamental_RemoveReference(
      (Fundamental_t*)&(pInstanceData->interfaceNodeBinaryOperatorNode));
  }

  return result;
}

static bool DefaultBinaryOperatorNode_SwitchInterface(
  DefaultBinaryOperatorNode_InterfaceNode_t* pInterfaceNode,
  wchar_t* pInterfaceName,
  Fundamental_t** ppResult)
{
  bool result = (pInterfaceNode != NULL) && (pInterfaceName != NULL);
  DefaultBinaryOperatorNode_InstanceData_t* pThis =
    pInterfaceNode->pInstanceData;

  if (result)
  {
    if ((wcscmp(pInterfaceName, FUNDAMENTAL_NAME) == 0) ||
        (wcscmp(pInterfaceName, NODE_NAME) == 0) ||
        (wcscmp(pInterfaceName, BINARY_OPERATOR_NODE_NAME) == 0))
    {
      if (ppResult != NULL)
      {
        *ppResult =
          (Fundamental_t*)&(pThis->interfaceNodeBinaryOperatorNode);
      }
    }
    else if (wcscmp(pInterfaceName, SCRIPTABLE_NAME) == 0)
    {
      if (ppResult != NULL)
      {
        *ppResult = (Fundamental_t*)&(pThis->interfaceNodeScriptable);
      }
    }
    else
    {
      result = false;
    }
  }

  if (ppResult != NULL)
  {
    if (result)
    {
      Fundamental_AddReference(*ppResult);
    }
    else
    {
      *ppResult = NULL;
    }
  }

  return result;
}

static void DefaultBinaryOperatorNode_AddReference(
  DefaultBinaryOperatorNode_InterfaceNode_t* pInterfaceNode)
{
  if (pInterfaceNode != NULL)
  {
    DefaultBinaryOperatorNode_InstanceData_t* pThis =
      pInterfaceNode->pInstanceData;
    pThis->referenceCount++;
  }
}

static void DefaultBinaryOperatorNode_RemoveReference(
  DefaultBinaryOperatorNode_InterfaceNode_t* pInterfaceNode)
{
  if (pInterfaceNode != NULL)
  {
    DefaultBinaryOperatorNode_InstanceData_t* pThis =
      pInterfaceNode->pInstanceData;

    pThis->referenceCount--;

    if (pThis->referenceCount == 0)
    {
      Node_RemoveReference(pThis->pLeftOperand);
      Node_RemoveReference(pThis->pRightOperand);
      free(pThis);
    }
  }
}
```

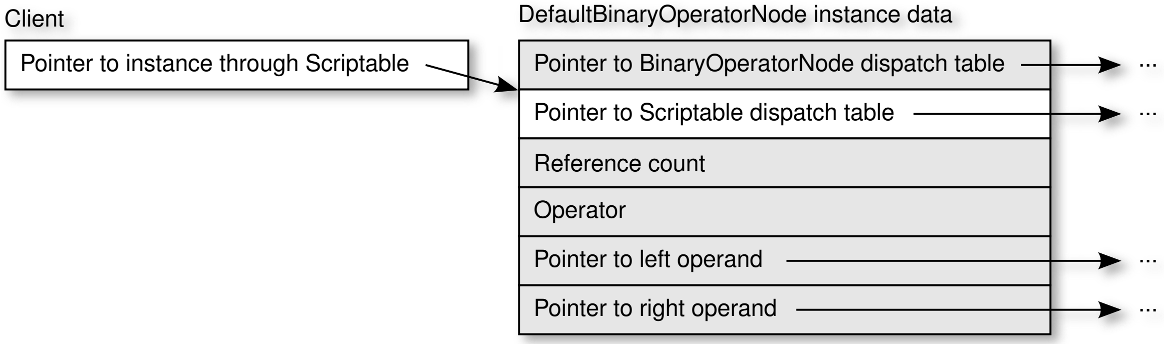
Client
Pointer to instance through Scriptable
DefaultBinaryOperatorNode instance data
Pointer to BinaryOperatorNode dispatch table
...
Pointer to Scriptable dispatch table
...
Reference count
Operator
Pointer to left operand
...
Pointer to right operand
...

Figure 4.3 Proposed instance data of `DefaultBinaryOperatorNode` objects

As noted earlier, the `BinaryOperatorNode` interface mandates a very specific memory layout of the memory referenced by pointers to objects implementing this interface (by way of `BinaryOperatorNode_t`, which appears in Listing 3.6 on page 40). The version of `DefaultBinaryOperatorNode` in Chapter 3 (see Listing 3.8 on page 42) puts a pointer to the dispatch table of the sole interface it implements first in its instance data, and as the memory layout of the instance data is consistent with that required by the binary standard, pointers that reference `DefaultBinaryOperatorNode` objects can simply point to the instance data.

For classes that implement multiple interfaces, multiple pointers to dispatch tables could be put first in the instance data, as seen in Figure 4.3 (as before, the shaded areas denote memory content that is not relevant to the binary standard, and which the implementation may use for any purpose). Client variables always point directly to the dispatch table that the client accesses the object through—in this figure, the client variable points directly to the `Scriptable` dispatch table. If the class only implements one interface, client pointers conveniently also point to the start of the instance data, allowing easy access to this data. Implementing multiple unrelated interfaces complicates matters, though.

As the interfaces implemented by a class are statically known, it is possible to deduce the address of the start of the instance data statically, and the layout presented in Figure 4.3 is thus fully adequate. Indeed, many compilers for object-oriented languages, such as C++, use a variation of this memory layout. A requirement is that a function is only accessed through one interface, as which interface a function is accessed through must be statically known in order to find the instance data.

However, for classes written directly in C, finding the start of the instance data is more involved. It would be possible to use pointer arithmetic, but for the sake of simplicity and clear code, an alternative approach will be used (which has the added benefit of allowing a single function to be accessed through many interfaces). This example uses *interface nodes*, one per implemented interface, that are compatible with types such as `BinaryOperatorNode_t`. Their first member points to the proper dispatch table, and their second (and last) member points back to the instance data, enabling functions, that now effectively take interface nodes as their first arguments, to retrieve a pointer to the instance data (which can be seen in Listing 4.3). This approach is depicted in Figure 4.4. Storing this data in a runtime-accessible data structure makes it easy for implementations to find the start of the instance data, as the offset to the address of the instance data is statically known.

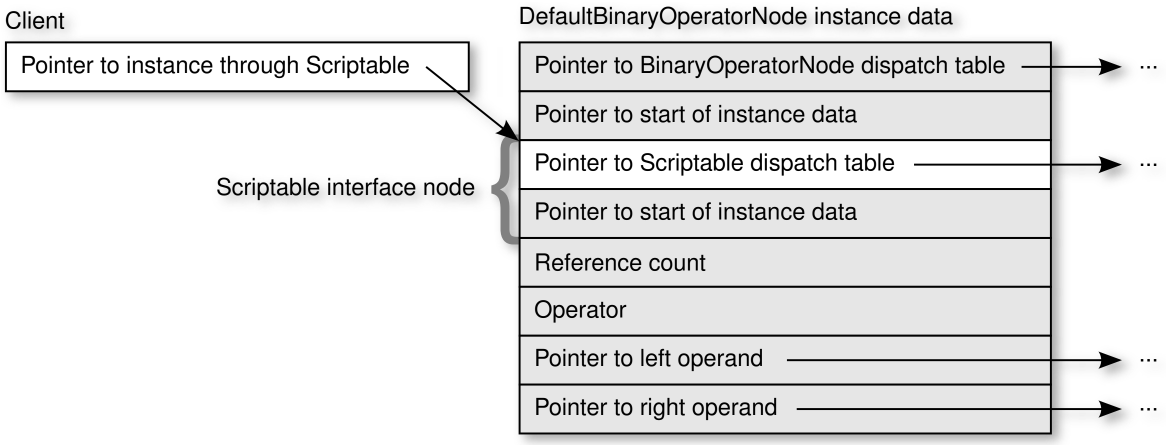
Client
Pointer to instance through Scriptable
DefaultBinaryOperatorNode instance data
Pointer to BinaryOperatorNode dispatch table
···
Pointer to start of instance data
Scriptable interface node
Pointer to Scriptable dispatch table
···
Pointer to start of instance data
Reference count
Operator
Pointer to left operand
···
Pointer to right operand
···

Figure 4.4 Revised instance data of `DefaultBinaryOperatorNode` objects

As `DefaultBinaryOperatorNode` now implements multiple interfaces, the functions implementing the operations of these interfaces are now interface-agnostic, meaning that the formal interface argument is now of the type `DefaultBinaryOperatorNode_InterfaceNode_t`. As a result, interfaces that share some of the same operations can share the same implementation (this is especially useful for the operations defined in `Fundamental`).[2] As a result, the dispatch tables need to use typecasting, making them somewhat harder on the eyes. As two separate interfaces are implemented, two dispatch tables are required as well; they are shown in Listing 4.4.

The new version of the constructor in Listing 4.3 is somewhat more involved, as it needs to initialize the interface nodes. (They are part of the instance data, and thus require no extra work to allocate and deallocate.) There is also some work involved in playing by the rules of reference counting; it makes copies of the left and right operand nodes (storing them as part of its instance data), and thus needs to manually add references to them. It uses the implementation of `Fundamental::SwitchInterface()` to return a proper reference through the desired interface, and must take care to remove the reference added to its own untyped reference before returning the reference returned by `Fundamental::SwitchInterface()`.

The implementation of `Fundamental::SwitchInterface()` is simple and functional, as are the implementations of the reference counting operations `Fundamental::AddReference()` and `Fundamental::RemoveReference()`. The simple strategy used in the implementation of `Fundamental::SwitchInterface()` does not scale well with a large number of implemented interfaces, though. It would benefit from the use of an efficient data structure, such as a hash table.

[2] Microsoft's COM considers interfaces, once published to the outside world, frozen and thus unchangeable. When new features need to be added, a new interface is created, which may only differ from the original by having one new operation. COM classes often implement several versions of an interface to maintain backwards compatibility. Having operation implementations that are interface-agnostic means that a class can very easily support multiple interfaces with overlapping operations by simply pointing to the same functions from the different dispatch tables.

Listing 4.4 Excerpt 2 from a revised version of DefaultBinaryOperatorNode.c

```c
/**
 * This is the dispatch table for the <code>BinaryOperatorNode</code>
 * interface implemented by this class.
 */
static const BinaryOperatorNode_DispatchTable_t
  gBinaryOperatorNodeDispatchTable =
{
  (bool (*)(BinaryOperatorNode_t*, wchar_t*, Fundamental_t**))
    DefaultBinaryOperatorNode_SwitchInterface,
  (void (*)(BinaryOperatorNode_t*))
    DefaultBinaryOperatorNode_AddReference,
  (void (*)(BinaryOperatorNode_t*))
    DefaultBinaryOperatorNode_RemoveReference,
  (bool (*)(BinaryOperatorNode_t*, bool*))
    DefaultBinaryOperatorNode_IsConstant,
  (bool (*)(BinaryOperatorNode_t*, unsigned int, unsigned int))
    DefaultBinaryOperatorNode_PrintDebugInformation,
  (bool (*)(BinaryOperatorNode_t*, BinaryOperator_t*))
    DefaultBinaryOperatorNode_Operator,
  (bool (*)(BinaryOperatorNode_t*, Node_t**))
    DefaultBinaryOperatorNode_LeftOperand,
  (bool (*)(BinaryOperatorNode_t*, Node_t**))
    DefaultBinaryOperatorNode_RightOperand
};

/**
 * This is the dispatch table for the <code>Scriptable</code>
 * interface implemented by this class.
 */
static const Scriptable_DispatchTable_t gScriptableDispatchTable =
{
  (bool (*)(Scriptable_t*, wchar_t*, Fundamental_t**))
    DefaultBinaryOperatorNode_SwitchInterface,
  (void (*)(Scriptable_t*))
    DefaultBinaryOperatorNode_AddReference,
  (void (*)(Scriptable_t*))
    DefaultBinaryOperatorNode_RemoveReference,
  (bool (*)(Scriptable_t*,
            wchar_t*,
            ScriptableReturnValue_t*,
            ScriptableArgumentType_t,
            ...))
    DefaultBinaryOperatorNode_InvokeOperation
};
```

4.2 Enabling very late binding

Very late binding makes it possible to access objects without having compile-time knowledge of them. This is especially useful for script hosts, but also for bridging solutions. In the context of component technology, bridging solutions make it possible for components written for one component model to be usable with code written for another. Like a script host, a bridge does not have compile-time knowledge of all interfaces it communicates with on behalf of other code, and as such needs to use very late binding.

In this chapter, very late binding is realized through late binding. The new interface that enables dynamically validated calls, `Scriptable`, is implemented by all classes that wish to be accessible from clients such as scripts. A script interpreter or a bridge has compile-time knowledge of this interface, and funnels all calls through it using late binding. It is up to the implementing class to check the validity of an incoming call (whether the given operation exists and the given arguments are of the correct types), call the proper operation (using late binding), and finally interpret and return the return value, if any.

The C header file of the `Scriptable` interface is displayed in Listing 4.5. It includes `ScriptableReturnValue_t` for return values, which relies on `ScriptableArgumentType_t`, shown in Listing 4.6.

Listing 4.5 Scriptable.h

```c
/**
 * @file
 *
 * This file contains an interface, <code>Scriptable</code>, which
 * allows invocations to be checked at runtime. In other words, it
 * facilitates very late binding. With late binding, the
 * implementations of interfaces are found at runtime, but invocations
 * are checked statically. An object implementing this interface may
 * be called from environments that do not allow invocations to be
 * checked statically, such as scripting languages.
 */

#ifndef INCLUSION_GUARD_SCRIPTABLE
#define INCLUSION_GUARD_SCRIPTABLE

#include <stdbool.h>
#include <wchar.h>
#include "Fundamental.h"
#include "ScriptableArgumentType.h"

/**
 * The name of this interface.
 */
#define SCRIPTABLE_NAME (L"se.polberger.components.Scriptable")

/* Convenience macros for the operations inherited from the
 * Fundamental interface:
 */
#define Scriptable_SwitchInterface(pScriptable,                     \
                                   pInterfaceName,                  \
                                   ppResult)                        \
  (pScriptable)->pDispatchTable->SwitchInterface(pScriptable,            \
                                                 pInterfaceName,         \
                                                 ppResult);
#define Scriptable_AddReference(pScriptable)                             \
  if (pScriptable != NULL)                                               \
  {                                                                      \
    (pScriptable)->pDispatchTable->AddReference(pScriptable);            \
  }
#define Scriptable_RemoveReference(pScriptable)                          \
  if (pScriptable != NULL)                                               \
  {                                                                      \
    (pScriptable)->pDispatchTable->RemoveReference(pScriptable);         \
  }

// Convenience macros for the operations defined in this interface:

/**
 * Invokes the specified operation with the given arguments. This
 * operation will fail if the underlying implementation does not
 * support the given operation, or if the argument list is not
 * correct. All arguments must be of the correct types.
 *
 * The <code>firstArgumentType</code> must be set to the type of the
 * first argument, or
 * <code>ScriptableArgumentType_NO_MORE_ARGUMENTS</code> if the called
 * operation accepts no arguments. It is followed by the data of the
 * first argument. If there are additional arguments, the next
 * argument must be set to the type of the next argument, followed by
 * its data, and so on. The last given argument to this operation must
 * be <code>ScriptableArgumentType_NO_MORE_ARGUMENTS</code>.
 *
 * @param[in] pScriptable
 *   the instance implementing this interface. Must not be
 *   <code>NULL</code>.
 * @param[in] pOperationName
 *   the name of the operation. This string is case-sensitive. Must
 *   not be <code>NULL</code.>
 * @param[out] pReturnValue
 *   a pointer to the variable which shall hold the return value
 *   returned from the called operation. This variable is normally
 *   allocated on the stack. If the return value holds a reference to
 *   an object implementing the <code>Scriptable</code> interface
 *   (that is, if <code>pReturnValue->type ==
 *   ScriptableArgumentType_SCRIPTABLE</code>), the caller is
 *   responsible for removing the reference that has been added.
 * @param[in] firstArgumentType
 *   the type of the first argument. This must be set to
 *   <code>ScriptableArgumentType_NO_MORE_ARGUMENTS</code> if the
 *   called operation does not accept any arguments.
 * @return
 *   <code>true</code> if the operation completes successfully,
 *   <code>false</code> otherwise.
 */
```

```
#define Scriptable_InvokeOperation(pScriptable,                          \
                                   pOperationName,                       \
                                   pReturnValue,                         \
                                   ...)                                  \
  (pScriptable)->pDispatchTable->InvokeOperation(pScriptable,            \
                                                 pOperationName,         \
                                                 pReturnValue,           \
                                                 __VA_ARGS__)

struct Scriptable_DispatchTable_s;

/**
 * Variables of this type may be used to represent objects
 * implementing this interface.
 */
typedef struct
{
  const struct Scriptable_DispatchTable_s* pDispatchTable;
} Scriptable_t;

struct ScriptableReturnValue_s;

/**
 * This is the dispatch table, or vtable, for this interface. It
 * represents an indirection that enables the implementation of an
 * interface to be bound to at runtime. Interfaces wishing to extend
 * this interface must include the complete contents of this structure
 * as the first members of their dispatch tables, changing the
 * <code>Scriptable_t</code> type to match their own. (Only single
 * interface inheritance is supported.)
 */
typedef struct Scriptable_DispatchTable_s
{
  // Operations inherited from the Fundamental interface:
  bool (*SwitchInterface)(Scriptable_t* pThis,
                          wchar_t* pInterfaceName,
                          Fundamental_t** ppResult);
  void (*AddReference)(Scriptable_t* pThis);
  void (*RemoveReference)(Scriptable_t* pThis);

  // Operations defined in the Scriptable interface:
  bool (*InvokeOperation)(Scriptable_t* pThis,
                          wchar_t* pOperationName,
                          struct ScriptableReturnValue_s* pReturnValue,
                          ScriptableArgumentType_t firstArgumentType,
                          ...);
} Scriptable_DispatchTable_t;

/**
 * This type represents a return value from a scriptable operation. It
 * is a tagged union, that is, a structure which contains the type of
 * its data, as well as a union representing the data.
 */
```

```
typedef struct ScriptableReturnValue_s
{
  /**
   * The type of the data stored in this tagged union (its
   * <em>tag</em>).
   */
  ScriptableArgumentType_t type;

  union
  {
    /**
     * Data corresponding to
     * <code>ScriptableArgumentType_INTEGER</code>.
     */
    unsigned int integerValue;

    /**
     * Data corresponding to
     * <code>ScriptableArgumentType_DOUBLE</code>.
     */
    double doubleValue;

    /**
     * Data corresponding to
     * <code>ScriptableArgumentType_BOOLEAN</code>.
     */
    bool booleanValue;

    /**
     * Data corresponding to
     * <code>ScriptableArgumentType_CHARACTER</code>.
     */
    wchar_t characterValue;

    /**
     * Data corresponding to
     * <code>ScriptableArgumentType_SCRIPTABLE</code>. If the return
     * value contains data of this type, the reference must be removed.
     * at some point.
     */
    Scriptable_t* pScriptableValue;
  } values;
} ScriptableReturnValue_t;

#endif // INCLUSION_GUARD_SCRIPTABLE
```

Listing 4.6 ScriptableArgumentType.h

```
/**
 * @file
 *
 * This file contains an enumerated type representing the different
 * types that may be passed to (and returned from) the
 * <code>Scriptable::InvokeOperation()</code> operation.
 */

#ifndef INCLUSION_GUARD_SCRIPTABLE_ARGUMENT_TYPE
#define INCLUSION_GUARD_SCRIPTABLE_ARGUMENT_TYPE

/**
 * This enumerated type represents the different types that may be
 * passed to (and returned from) the
 * <code>Scriptable::InvokeOperation()</code> operation.
 */
typedef enum
{
  /**
   * The type is invalid.
   */
  ScriptableArgumentType_INVALID,

  /**
   * There is no type. This is used for non-existent return values,
   * and may not be used for arguments to an operation.
   */
  ScriptableArgumentType_VOID,

  /**
   * The type represents an unsigned integer (an <code>unsigned
   * int</code>).
   */
  ScriptableArgumentType_INTEGER,

  /**
   * The type represents a double-precision floating-point value
   * conforming to IEEE 754 (a <code>double</code>).
   */
  ScriptableArgumentType_DOUBLE,

  /**
   * The type represents a boolean value (a <code>bool</code>).
   */
  ScriptableArgumentType_BOOLEAN,

  /**
   * The type represents a character (a <code>wchar_t</code>).
   */
  ScriptableArgumentType_CHARACTER,

  /**
   * The type represents an arbitrary object which implements the
   * <code>Scriptable</code> interface.
   */
  ScriptableArgumentType_SCRIPTABLE,

  /**
```

```
   * This enumerator signifies that no more arguments are expected in
   * a call to <code>Scriptable::InvokeOperation()</code>.
   */
  ScriptableArgumentType_NO_MORE_ARGUMENTS
} ScriptableArgumentType_t;

#endif // INCLUSION_GUARD_SCRIPTABLE_ARGUMENT_TYPE
```

`Scriptable` has only one operation, the “meta operation” `InvokeOperation()`. A client calls this operation to invoke a named operation offered by the class. It passes the name of the operation to invoke, followed by a pointer to a (preferably stack-allocated) tagged union which is to hold the return value after the commencement of the call, and finally the arguments that are to be passed to the invoked operation. The arguments are given in pairs: the first member of the pair denotes the type of the argument, and the second member the actual data. `ScriptableArgumentType_NO_MORE_ARGUMENTS` must be the last argument. Listing 4.7 shows an excerpt from a program testing late and very late binding by calling `Node::PrintDebugInformation()` using both invocation mechanisms.

Listing 4.7 Excerpt from NodeTest.c

```
  if (result)
  {
    printf("\n");
    printf("Tree (using late binding):\n");
    result = Node_PrintDebugInformation(pRootNode, 0, 2);
  }

  if (result)
  {
    printf("\n");
    printf("Tree (using very late binding):\n");

    Scriptable_t* pScriptableRootNode = NULL;
    ScriptableReturnValue_t returnValue;

    result =
      Scriptable_SwitchInterface(pRootNode,
                                 SCRIPTABLE_NAME,
                                 (Fundamental_t**)&pScriptableRootNode);

    if (result)
    {
      result = Scriptable_InvokeOperation(
        pScriptableRootNode,
        L"PrintDebugInformation",
        &returnValue,
        ScriptableArgumentType_INTEGER,
        0,
        ScriptableArgumentType_INTEGER,
        2,
        ScriptableArgumentType_NO_MORE_ARGUMENTS);
    }
    if (result)
    {
      result = (returnValue.type == ScriptableArgumentType_VOID);
    }

    /* While we don't expect the return type to be of type Scriptable,
     * we are contractually obligated to remove the reference if this
     * type is returned.
     */
    if (returnValue.type == ScriptableArgumentType_SCRIPTABLE)
    {
      Scriptable_RemoveReference(returnValue.values.pScriptableValue);
    }

    Scriptable_RemoveReference(pScriptableRootNode);
```

The supported types, for both input arguments and return values, are unsigned integers, double-precision floating point values, boolean values, wide characters, as well as arbitrary objects implementing the `Scriptable` interface. There is no type for strings, as the most prudent way to represent a string is arguably as an object, a strategy which ensures that memory will be properly managed through reference counting. A new interface, `String`, could be introduced, and provided that classes implementing this interface also implement `Scriptable`, strings would be available through very late binding. (Though it would likely be more appropriate for a script host to have compile-time knowledge of the `String` interface and map native strings in the scripting language to the `String` interface using late binding.)

`DefaultBinaryOperatorNode` implements the new interface in the most straightforward way possible. Its performance is not optimal; using a data structure such as a hash table would greatly speed up the string comparisons. The implementation is shown in Listing 4.8.

Listing 4.8 Excerpt 3 from a revised version of DefaultBinaryOperatorNode.c

```
static bool DefaultBinaryOperatorNode_InvokeOperation(
  DefaultBinaryOperatorNode_InterfaceNode_t* pInterfaceNode,
  wchar_t* pOperationName,
  ScriptableReturnValue_t* pReturnValue,
  ScriptableArgumentType_t firstArgumentType,
  ...)
{
  bool result =
    (pInterfaceNode != NULL) &&
    (pOperationName != NULL) &&
    (pReturnValue != NULL) &&
    (firstArgumentType != ScriptableArgumentType_INVALID) &&
    (firstArgumentType != ScriptableArgumentType_VOID);
  BinaryOperatorNode_t* pBinaryOperatorNode =
    (BinaryOperatorNode_t*)
    &(pInterfaceNode->pInstanceData->interfaceNodeBinaryOperatorNode);

  va_list arguments;
  va_start(arguments, firstArgumentType);

  if (result)
  {
```

```
if (wcscmp(pOperationName, SCRIPTABLE_OPERATION_IS_CONSTANT) == 0)
{
  bool isConstant = false;

  // This operation takes no input arguments.
  result =
    (firstArgumentType == ScriptableArgumentType_NO_MORE_ARGUMENTS);

  if (result)
  {
    result = BinaryOperatorNode_IsConstant(pBinaryOperatorNode,
                                           &isConstant);
  }

  if (result)
  {
    pReturnValue->type = ScriptableArgumentType_BOOLEAN;
    pReturnValue->values.booleanValue = isConstant;
  }
}
else if (wcscmp(pOperationName,
                SCRIPTABLE_OPERATION_PRINT_DEBUG_INFORMATION) == 0)
{
  ScriptableArgumentType_t lastType = firstArgumentType;
  unsigned int argumentCount = 0;
  unsigned int startPosition = 0;
  unsigned int indentationSize = 0;

  // This operation takes two arguments, both integers.
  result = (lastType != ScriptableArgumentType_NO_MORE_ARGUMENTS);

  while (result &&
         (lastType != ScriptableArgumentType_NO_MORE_ARGUMENTS))
  {
    argumentCount++;
    result = argumentCount <= 2;

    if (result)
    {
      result = (lastType == ScriptableArgumentType_INTEGER);
    }

    if (result)
    {
      switch (argumentCount)
      {
        case 1:
          startPosition = va_arg(arguments, unsigned int);
          break;

        case 2:
          indentationSize = va_arg(arguments, unsigned int);
          break;
        default:
          result = false;
          break;
      }
    }

    if (result)
    {
      lastType = va_arg(arguments, ScriptableArgumentType_t);
    }
  }

  if (result)
  {
    result =
      BinaryOperatorNode_PrintDebugInformation(pBinaryOperatorNode,
                                                startPosition,
                                                indentationSize);
  }

  if (result)
  {
    pReturnValue->type = ScriptableArgumentType_VOID;
  }
}
else if (wcscmp(pOperationName, SCRIPTABLE_OPERATION_OPERATOR) == 0)
{
  BinaryOperator_t operator = BinaryOperator_UNDEFINED;

  // This operation takes no input arguments.
  result =
    (firstArgumentType == ScriptableArgumentType_NO_MORE_ARGUMENTS);

  if (result)
  {
    result = BinaryOperatorNode_Operator(pBinaryOperatorNode,
                                         &operator);
  }

  if (result)
  {
    pReturnValue->type = ScriptableArgumentType_INTEGER;
    pReturnValue->values.integerValue = (unsigned int)operator;
  }
}
else if (wcscmp(pOperationName,
                SCRIPTABLE_OPERATION_LEFT_OPERAND) == 0)
{
  Node_t* pNode = NULL;
  Scriptable_t* pScriptableNode = NULL;

  // This operation takes no input arguments.
```

```
      result =
        (firstArgumentType == ScriptableArgumentType_NO_MORE_ARGUMENTS);

      if (result)
      {
        result = BinaryOperatorNode_LeftOperand(pBinaryOperatorNode,
                                                &pNode);
      }

      if (result)
      {
        /* The return value is an object accessed through the Node
         * interface. We can only return this value if the class
         * implementing this interface also supports the Scriptable
         * interface, and we thus need to test for this at runtime.
         */
        result = Node_SwitchInterface(pNode,
                                      SCRIPTABLE_NAME,
                                      (Fundamental_t**)&pScriptableNode);
      }

      if (result)
      {
        pReturnValue->type = ScriptableArgumentType_SCRIPTABLE;
        pReturnValue->values.pScriptableValue = pScriptableNode;
      }

      Node_RemoveReference(pNode);
    }
    else if (wcscmp(pOperationName,
                    SCRIPTABLE_OPERATION_RIGHT_OPERAND) == 0)
    {
      /* This code is very similar to the one for handling the left
       * operand, and has therefore been omitted from this code
       * listing.
       */
    }
    else
    {
      result = false;
    }
  }

  va_end(arguments);

  if ((!result) && (pReturnValue != NULL))
  {
    pReturnValue->type = ScriptableArgumentType_INVALID;
  }

  return result;
}
```

4.3 Object-oriented omissions

The object model developed in this chapter and in Chapter 3 is fairly complete. Objects are prevented from interfering with one another, as they may only access the state of another object through the interfaces it implements (realizing encapsulation).[3] Also, objects can have many "personalities" by implementing multiple interfaces (realizing polymorphism, which dynamic dispatch makes possible). A few aspects that are often part of object models have been omitted, though, and these aspects are discussed in this section.

4.3.1 Class interface

A class traditionally provides both an implementation and an interface. It is the interface that enables a class to be used as a type. Interfaces wedded to a particular class are known as *class interfaces*, and are the sole means of accessing objects in languages with no concept of freestanding interfaces. Many modern object-oriented languages, such as Java and C#, are hybrids in that they support both freestanding and class interfaces. A class interface in such languages contains all operations found in the freestanding interfaces implemented by the class, in addition to operations only accessible through the class interface.

This object model is unorthodox in that there are no class interfaces, and classes can thus not be used directly as types.[4] The only means of accessing a class is through one of the freestanding interfaces it implements, which means that classes in this object model have been reduced to pure implementation entities. This means that only virtual operations are supported, thus necessitating the use of dynamic dispatch.

4.3.2 Implementation inheritance

Implementation inheritance is traditionally considered one of the three pillars of object-oriented programming, along with encapsulation and polymorphism. This supposed third pillar has been omitted from this object model.

Implementing support for single implementation inheritance would be straight-forward, and Stroustrup (1999) demonstrates that supporting multiple implementation inheritance adds very little in terms of complexity, runtime cost and memory overhead. Making implementation inheritance part of the binary standard of this object model, however, allowing classes written in different languages to extend one another, would add considerable complexity.

There are reasons to forego implementation inheritance completely. Snyder noted that implementation inheritance breaks encapsulation as early as 1986. Szyperski et al. (2002:115) offer a critique that centers on the tight dependency between classes and their ancestor classes, called the *fragile base class problem.* The syntactic variant of the problem refers to the inability to modify a class without recompiling all descendant classes and dependent clients, and is solely concerned with binary compatibility (as compiled-in offsets are no longer correct when the base class is modified). This problem was solved by IBM's System Object Model (SOM), by initializing dispatch tables at load-time.

[3] Malicious code can, of course, easily wreak havoc with the internal state of any accessible object, but this is true for all object models implemented in native code, including C++. The point is, though, that the object model, properly used, prevents such access.

[4] Interfaces can, of course, be constructed that are identical to what a class interface would look like in Java, and be designated as such in the documentation.

The semantic variant of the problem is more interesting, and is concerned with changes to the behavior of ancestor classes that a descendant class cannot cope with. If a class overrides selected (virtual) operations of an ancestor class, it may become dependent on the ancestor class calling these operations, perhaps even in a certain order. The dependence of the descendant class on the behavior of the ancestor class is not regulated by the formal (syntactic) contract between the two classes. The tight coupling that results makes evolution of the ancestor classes difficult or impossible.

The recommended way to avoid implementation inheritance is to use *forwarding*, sometimes known as *delegation*, which entails forwarding calls to internally-held objects.[5] Forwarding calls to an internally held reference works just as well for code reuse as inheriting an implementation (with the downside that most languages require significantly more boiler-plate code to be written). It does not allow for the same level of customization of another class as that afforded by overriding operations of an ancestor class, though. Support for customization must be built into the ancestor class, and not patched in, as is done when virtual operations are overridden.

4.3.3 Access specifiers

This object model does not support making parts of the instance data accessible to code outside the class (such fields are often designated with a `public` access specifier in object-oriented languages). Such support could easily be added (by exposing a structure containing the public fields, and adding an operation to the `Fundamental` interface returning a pointer to this structure), but it is probably preferable to eschew exposing parts of the instance data for reasons of encapsulation.

All operations that are part of a dispatch table are per definition publicly accessible. Private operations are easily realized by simply not exposing them in a dispatch table (and declaring them `static`). (Private operations are always bound to statically, as they may only be accessed by operations that belong to their own class.)

Object-oriented languages often support a `protected` access specifier that exposes a field or operation only to descendant classes. As this object model does not support implementation inheritance, there is no need for such an access specifier.

4.3.4 Multiple interface inheritance

The interfaces found in this chapter descend directly from one or zero other interfaces. Multiple interface inheritance involves descending from multiple interfaces, and is somewhat more involved to implement. This feature is useful as it allows for greater design expressiveness—an interface can mandate that classes implementing it also implement a host of other related interfaces.

With single interface inheritance, no matter how many ancestor interfaces an interface has, classes only have to provide one dispatch table in order to successfully implement it. `Scriptable`, which descends directly from `Fundamental`, provides only one dispatch table type, `Scriptable_DispatchTable_t` (which is shown in Listing 4.5 on page 62). Such dispatch tables also work for `Fundamental` (whose dispatch table type appears in Listing 4.1 on page 51), because the members of `Fundamental_DispatchTable_t` appear before the new members introduced in `Scriptable_DispatchTable_t`.

[5] Embarcadero's Delphi programming language includes a language feature that makes forwarding less verbose than having to manually forward every call. This feature is described in the footnote on page 82.

With multiple interface inheritance, things are more involved. The approach taken by Stroustrup (1999) is to mandate that classes that implement interfaces that descend from multiple interfaces provide multiple dispatch tables. If interface `Z` descends from both `X` and `Y`, two dispatch tables would have to be provided, one containing the members from the dispatch tables of `X` and `Z`, and the other containing the members of `Y` and `Z`. Had support for multiple interface inheritance been supported, `Fundamental::SwitchInterface()` would use different dispatch tables depending on which interface was sought.

4.4 Moving toward component technology

The object model presented in this chapter is a binary standard. As such, it is concerned with the layout of memory pointed to by object references, as well as the layout of dispatch tables. Clients making use of objects conforming to this object model also need information on what calling convention to use, as well as information on the type system, which unambiguously specifies the memory representation of types used by the object model.

As a binary standard, the C code that realizes classes is not normative; it merely plays by the rules of the standard. This is precisely how component models based on binary standards, like Microsoft's COM, work. In fact, this object model could, with a few additions, be expanded to the point where it would qualify as a component model. These additions are discussed in this section. (The component model sketched here borrows liberally from the playbook of COM.)

4.4.1 Factories

As things stand, classes such as `DefaultBinaryOperatorNode` can only be instantiated by calling a C function specific to the class that is to be instantiated. Component models typically strive to decouple clients from the classes they instantiate, which makes it possible to instantiate classes without having compile-time knowledge of them. This property is especially useful for scripting languages, as script hosts need to instantiate classes on behalf of scripts they are interpreting, classes they have not been compiled against.

To realize this, a component model implementation needs to provide a means of instantiating classes as part of a runtime system that all users of the component model are linked against. This can take the form of a standard function called using procedural calling conventions, or the form of an object implementing a system-provided interface. In order to instantiate classes, this function or interface operation needs to be given an argument identifying the class to be instantiated. As such, classes must be given runtime names, in the same vein as the runtime names for interfaces presented in this chapter.

Different languages targeting the component model sketched here may use different memory allocation strategies when instantiating objects. `DefaultBinaryOperatorNode`, for instance, uses the `malloc()` function, part of the C standard library, to allocate memory for its object instance. As a component model implementation cannot be privy to these details, it delegates the work of instantiating objects to factories (as noted in section 2.2.5). A factory could aptly be represented as an object implementing an interface with only one operation, `CreateInstance()`.

A component model implementation needs to be able to map runtime names of classes to the factories that are capable of instantiating said classes, and as such needs access to a data store that holds this information. In a statically linked system (that is, one that does not support components per se), a data store in this vein would only need to map runtime names to addresses of factory objects. A system supporting true components would instead need to locate the component housing the class to instantiate, and get an appropriate factory from this component.

4.4.2 Code generation

Writing the low-level C code that realizes classes and interfaces that adhere to this object model is arguably tedious and error-prone. The many languages that improve on C by adding language-level support for object-orientation, such as C++ and Objective-C, are a testament to the usefulness of having support for object-orientation at the language level.

Much of this low-level C code could be generated, though, provided that the relevant classes and interfaces are described in an interface description language. An IDL compiler could completely generate the interface files, including the access macros and the client variable and dispatch table types. Class implementations could be generated as two files, one housing dispatch tables and runtime names, and one containing the operation implementations. Only the latter file would need to be manually edited to provide the domain-specific functionality. Implementations of all `Fundamental` operations could also be generated, as well as an implementation of `Scriptable::InvokeOperation()`, if support for very late binding is desired. Factories could also be automatically generated, as could proxies used for inter-process and inter-machine communication, if support for location-transparent invocations is added to this component model.

4.4.3 Runtime type information

Many modern languages provide the ability for programs to examine and even modify the behavior of themselves. Java, for instance, has elaborate support for *reflection*, enabling programs to locate classes, iterate over their methods, and invoke methods—all at runtime. With such elaborate support for reflection built into the platform itself, supporting scripting languages is trivial. There would be no need for a `Scriptable` interface; a script interpreter would use the reflection services of the platform to reflectively verify the correctness of invocations before dispatching them.

Providing this level of support for reflective operations requires that type information normally only available at compile-time is available at runtime. In the solution presented in Listing 4.8, some of this data is indeed present, embedded in the imperative code that makes up the implementation of `Scriptable::InvokeOperation()`. In order to provide a fuller reflection service, such data would need to be available as part of a generic data structure. With an interface description language and an IDL compiler, this data structure could be populated by generated code.

If every class makes type information on itself available at runtime, perhaps by adding an operation to that effect to the `Fundamental` interface, it is possible to write class-agnostic implementations of `Fundamental::SwitchInterface()` and `Scriptable::InvokeOperation()` that could be shipped as part of the runtime system of a component model implementation. The former operation would consult the type information to see if a requested interface was implemented by the object, and if so, return a proper reference. The latter operation would check the validity of invocations against the type information before invoking calls. Relying on class-agnostic, generic implementations that use runtime-accessible type information saves on code size compared to generating class-specific code.

A class-agnostic implementation of `Scriptable::InvokeOperation()` would be somewhat more complex to write than simply generating class-specific code, though. In particular, a class-specific implementation can perform a normal C function call to invoke an operation after verifying the correctness of the call, whereas a class-agnostic implementation would not have the luxury of having the C compiler generate the machine instructions performing the invocation. Instead, a class-agnostic implementation would manually have to construct a stack frame to be pushed onto the call stack, and this part would need to be written in architecture-specific assembly language.

4.4.4 Software components

The last major piece of the puzzle are the software components themselves. The most reasonable way of implementing components is to piggyback on shared libraries (dynamically linked libraries), that already provide many of the services needed by components.

Most software is implicitly linked with shared libraries (also known as “load-time dynamic linking”). A program declaratively states which libraries it depends on, and the dynamic linker of the operating system loads the dependent libraries into the address space of the program at load-time, that is, before the program starts executing. As a shared library is potentially loaded at an address which is not known at compile-time, a runtime structure, called a *jump table*, is often used that holds the addresses to a shared library’s exported functions (Hunt and Scott 1999). It is initialized by the dynamic linker at load-time, and is consulted before functions in the shared library are called.[6]

Functions that realize object operations in this chapter are not exported—indeed, as they are declared `static`, their symbol names are not even visible outside their compilation units. Instead, they are accessed through dispatch tables. When a class is put in a shared library, dispatch tables do double duty as jump tables, as well as serving their traditional role of decoupling implementation from interface.

Shared libraries housing components should export one function though, one that makes it possible to instantiate objects. This function should take the runtime name of a class as an argument, and return a factory object capable of instantiating this class. Components must be loaded using explicit linking (also known as “runtime dynamic linking”), as all components use the same symbol name for this function. When a shared library is loaded explicitly, operating systems universally return a handle that can be used to refer to the library at runtime.[7] In this context, this handle can be considered the runtime identity of the component, which makes it possible to differentiate between components at runtime.

Components should also provide version information on themselves, and state what other components they depend on. The runtime library of the component model discussed here should take this information into account when loading a component, to make sure that all dependencies are met.

With that, the broad outlines of a component model implementation have been sketched.

[6] Some systems, notably Windows, modify the executable code directly instead of using jump tables.

[7] POSIX-compliant systems, such as Linux and other Unix-like systems, provide the `dlopen()`, `dlclose()` and `dlsym()` functions to link in shared libraries explicitly at runtime (`dlopen()` and `dlclose()` load and unload a shared library, respectively, and `dlsym()` returns the address of the function associated with a given symbol name). The equivalent functions under Windows are `LoadLibrary()`, `FreeLibrary()` and `GetProcAddress()`, which roughly correspond to the aforementioned POSIX functions, in that order.

CHAPTER 5

Ways of the industry

For the most part, component technology grew out of industry, not academia, and evolved from multiple directions. In the enterprise realm, the Open Group's Distributed Computing Environment (DCE) was developed in the early 1990s, and provided the infrastructure for creating distributed applications. It was the first solution to use a formalized interface description language. The IDL dialect of DCE was used to automatically generate client-side and server-side proxies, hiding the low-level machinery used for inter-process and inter-machine communication and thus making distributed computing vastly simpler to implement (Open Software Foundation 1995; Hludzinski 1998). DCE only provided for remote procedure calls with no object semantics, though. This was rectified by the Object Management Group's Common Object Request Broker Architecture (OMG's CORBA), which helped heterogeneous object-oriented systems interoperate.

Microsoft's efforts started on the desktop, and grew out of an effort to make its office productivity applications interoperate better. A technology dubbed Object Linking and Embedding (OLE) enabled documents created in one application to link to or embed documents created in other applications. OLE made it possible for, say, a presentation slide to contain an embedded spreadsheet, which could be edited "in-place" by double-clicking it (without leaving the presentation software). The component technology that resulted was the Component Object Model (COM), whose later incarnations competed with CORBA in the enterprise space. Other organizations also developed technology for intermingling different kinds of media in a single document (such documents are known as *compound documents*). Apple based its (now demised) competing technology OpenDoc on International Business Machines's System Object Model (IBM's SOM) (Alger 1994).

Again on the desktop, Microsoft enjoyed early success with Visual Basic and its support for third-party software made available as "custom controls." Visual Basic later based its component technology on COM. Borland's Delphi product was compatible with components built for Visual Basic, but also sported its own object and component models, and a fully compiled, strongly-typed language.

Many of the more popular current component models for the desktop and the enterprise are built on platforms based on capable virtual machines, considerably simplifying component technology. Sun's Java technology, as well as Microsoft's COM successor, .NET, belong to this category.

5.1 Visual Basic

Microsoft's Visual Basic, first introduced in 1991, married the BASIC programming language with a productive integrated development environment. This environment allowed for the visual construction of applications by simply dragging visual "controls" from a toolbox to a form, customizing said controls using attributes (such as the caption of a push button), and attaching BASIC code to events.

The first versions of Visual Basic did not compile to native code, and instead relied on an interpreter which negatively impacted performance. Controls were typically written in C or C++ and packaged as Visual Basic Extensions (VBX), which can be regarded as the components of early Visual Basic. A VBX file was a Windows shared library (a Dynamic-Link Library, or DLL) which used a Visual Basic-specific application programming interface to communicate with its host, and was expected to export a well-defined set of symbols specific to Visual Basic. VBX controls were thus tightly coupled to Microsoft's product.

Early versions of Visual Basic perfectly exemplified the division between component assemblers and component writers, advocated by some proponents of component technology (discussed on page 4). Component assemblers would compose components using the easy-to-learn BASIC language and Visual Basic form designer, and component writers would create components using comparatively complex native languages such as C or C++. The performance cost of interpreting BASIC code is easy to bear in this line of thinking, as performance-critical code is expected to be part of controls written in native code.[1]

Visual Basic gave rise to a successful component market. Controls ranged from simple visual controls, such as sliders, to full-blown spreadsheet engines and database connectivity tools (Szyperski et al. 2002:351). Despite the lack of a formal specification for Visual Basic components, several vendors created competing products that could act as their hosts (Udell 1994).

The first versions of Visual Basic were the epitome of an environment sporting a first-generation component model, as discussed in Chapter 2. Later versions gained native code generation, and the ability to not only use COM components, but also author them. In fact, 32-bit versions of Visual Basic did not use the VBX component model, and instead based their components on COM, which were dubbed OLE custom controls (OCX) (Szyperski et al. 2002:351). A variation of this technology later came to be known as ActiveX controls, which also served as the plug-in technology for Microsoft's web browser, Internet Explorer. The current version of Visual Basic has little in common with its predecessors and is fully based on .NET.

5.2 COM

The Component Object Model (COM) grew out of Microsoft's efforts to make the various parts of its Office productivity suite work together. The first version of the Object Linking and Embedding (OLE) technology allowed one application, such as Microsoft Word, to embed the content of another, such as Microsoft Excel, thus realizing compound documents. The second version of OLE added things like in-place editing and drag-and-drop. Whereas OLE 1.0 was a

[1] Visual Basic's interpreter did not interpret BASIC code, it interpreted *p-code* (pseudo code), which can be likened to Java bytecode. Visual Basic 5.0, introduced in 1997, gained the ability to compile to native code. Some versions of Microsoft's 16-bit C++ compiler were also able to produce p-code. The chief selling point of p-code was not portability, but smaller executable file sizes.

self-contained technology, Microsoft opted to separate the object and component models from OLE when building OLE 2.0, as it realized that the technology built for OLE could be useful in other contexts as well. Tony Williams, one of the architects of COM at Microsoft, puts it thus: "We decided to bite the bullet and make a much more formalized model of what is an object, what does it mean to have interfaces, what does it mean to negotiate over them—and [...] that became COM" (Microsoft 2006). While COM and OLE were both built by the same team, Microsoft kept the distinction between them very clear. OLE 2.0 was layered on top of COM, and consisted of a large number of interfaces and a small runtime system.

COM has evolved considerably since then. The COM runtime system ships with all modern Windows versions, and a large number of Windows services are accessed using COM interfaces. COM has also proven popular as a vehicle for realizing component technology for third-party developers. The rich ecosystem that has grown up around ActiveX components, which are based on COM, is a case in point (Szyperski et al. 2002:26). COM may be used to realize distributed computing through Distributed COM (DCOM), and enterprise services (such as the ones enumerated in section 1.7) through COM+ (Bukovics 2006).

COM has had a profound influence on the industry. While the original runtime system powers a large number of applications and services on the Windows platform, it is the ideas that underlie COM that have had the greatest impact. Many component models are more or less faithful replicas of COM. Despite the Windows-centric nature of COM,[2] the technology is widely available on other platforms, reimplemented by vendors other than Microsoft. Mozilla's products, such as the Firefox web browser, use a COM-compatible technology known as XPCOM ("XP" stands for "cross-platform") (Turner and Oeschger 2003).[3] Sun's OpenOffice.org productivity suite uses the COM-inspired UNO component model to enable OLE-like features, such as embedding, in-place activation and scripting using the BASIC language (Sun Microsystems 2007).

In the embedded realm, many component models are heavily influenced by COM. Philips's and Samsung's Universal Home API (UHAPI), for electronics appliances in the home, borrows heavily from COM with its uhCOM technology. The Symbian operating system for mobile handsets uses a COM-inspired component model called ECom (Symbian Foundation 2008). ABB uses COM-like technology for their programmable controllers to increase the modularity of their codebase (Lüders et al. 2005).

5.2.1 Technical foundation

COM is both an object model and a component model—components encapsulate instantiable classes, which implement interfaces, through which objects communicate. As a binary standard, it standardizes aspects of components and objects that are important to the correct functioning of COM. It is agnostic to the implementation language used to write classes, as long as the binary standard is adhered to.

Through its binary standard, COM standardizes the access mechanism for objects by mandating a specific memory layout, calling convention and type system. As interfaces are the sole means of accessing COM objects, COM is said to be a binary standard for

[2] There have been attempts to port COM to other platforms, notably Unix and the Apple Macintosh. Such efforts have gained little traction, though (Szyperski et al. 2002:330).

[3] XPCOM is interesting in that components are written in C++, user interfaces described in an XML-based language and the glue between them written in the scripting language JavaScript (again, exemplifying the division between component assemblers and component builders).

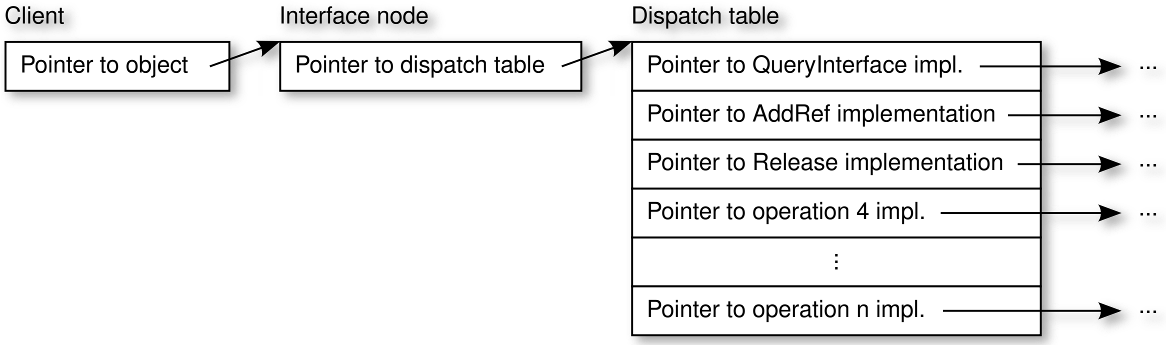
Client
Interface node
Dispatch table
Pointer to object
Pointer to dispatch table
Pointer to QueryInterface impl.
Pointer to AddRef implementation
Pointer to Release implementation
Pointer to operation 4 impl.
⋮
Pointer to operation n impl.
...

Figure 5.1 Memory layout of COM interfaces

interfaces. The memory layout mandated by COM is notably compatible with pure virtual C++ classes, as produced by Microsoft's own C++ compiler.[4] As a result, COM goes some way toward standardizing a C++ application binary interface (ABI) on the Windows platform, making code produced by different C++ compilers compatible, at least as far as COM is concerned.[5] Vendors of compilers for languages other than C++ need to adhere to the standard as well, if their language is to be compatible with COM. Embarcadero's Delphi integrated development environment, whose Delphi programming language is a variation of Object Pascal, fully adheres to the COM binary standard by producing COM-conformant objects (Calvert 1999:381). COM thus successfully creates a standard that enables disparate object-oriented languages to communicate without losing their object semantics, and does so by only standardizing the absolute minimum required to ensure binary interoperability.

The binary standard of COM is similar, but not identical, to the binary standard developed in Chapter 4. Like this binary standard, COM does not support implementation inheritance, which can be seen as a feature (as argued on page 70). A client variable points to a memory area whose first member points to a dispatch table, the fields of which point to the actual implementation (Szyperski et al. 2002:330). Figure 5.1 depicts this visually. The first argument to a function that serves as the implementation of a COM operation must be a *this* pointer, which again is consistent with Chapter 4. This allows COM to exhibit true object characteristics.[6]

[4] If a compiler produces classes that are not compatible with COM's binary standard, these classes can not be used as the direct implementation vehicles for COM classes. However, COM is fully accessible to any program compiled to native code that gives direct access to memory, such as programs written in C. A COM class written in C does not look all that different from the classes presented in Chapter 4.

[5] There is much more to C++ binary compatibility than accessing objects created from pure virtual classes. This includes exception handling, runtime type information, name mangling and (possibly multiple) implementation inheritance (Clamage 2002). COM sidesteps many of these issues—return values are used instead of exceptions, implementation inheritance is not supported and name mangling above that standardized by C is not needed, as COM's binary standard is based on dispatch tables. Also, COM provides its own standards for runtime type information, in the form of `IUnknown::QueryInterface()` and type libraries.

[6] On 32-bit machines, COM uses the `stdcall` calling convention instead of the `thiscall` calling convention normally used by Microsoft's C++ compiler. The latter calling convention passes the *this* pointer via a register and not on the call stack, whereas `stdcall` passes all arguments on the stack. C can easily support COM, as most C implementations for Windows support the `stdcall` calling convention. As the callee is responsible for cleaning the stack when `stdcall` is used, this calling convention (and thus COM) cannot support variadic arguments (operations that take a caller-determined number of arguments).

A COM class may implement any number of interfaces. All interfaces directly descend from one other interface, except `IUnknown`, the root of the interface inheritance hierarchy (by convention, all compile-time interface names start with "I"). `IUnknown` is, for most intents and purposes, identical to the `Fundamental` interface introduced in Chapter 4. The COM equivalent to the `Fundamental::SwitchInterface()` operation is `IUnknown::QueryInterface()`, and `Fundamental::AddReference()` and `Fundamental::RemoveReference()` correspond to `IUnknown::AddRef()` and `IUnknown::Release()`, respectively.

COM uses reference counting to manage memory. Objects need not be reference counted in their entirety—each interface implemented by an object can be separately reference counted. This feature is known as *tear-off interfaces*, and can be used by an object to initialize and destroy resources on a per-interface basis, thus conserving resources (Szyperski et al. 2002:334).

Runtime names need to be assigned to a large number of different COM entities, including classes and interfaces. COM uses Universally Unique Identifiers (UUIDs) for this purpose, also referred to as Globally Unique Identifiers (GUIDs) by Microsoft. A UUID is a 128-bit number which has a very high probability of being globally unique. A textual representation of a UUID can look as follows: "7a3fc5d3-f79a-4de5-827d-d0f5619a4c99." The Windows Registry serves as the data store that maps UUIDs to components (shared libraries for objects that run in-process and executable files for objects that run out-of-process). (Recent Windows versions support in-process COM components that are not globally accessible, and thus do not need to be stored in the Registry (Templin 2005).)

Operations in COM interfaces are expected to provide error information in the form of integer return values known as `HRESULT`. True return values are provided as output arguments. A `HRESULT` value is a 32-bit integer value divided into a number of fields.

To the extent that COM supports versioning, it does so through avoidance. An interface UUID identifies not only an interface, but also its version, thus requiring that interfaces, once published, are never changed. A class can easily support multiple versions of an interface by implementing all interfaces corresponding to the different versions. Newer clients use `IUnknown::QueryInterface()` to query for a newer version, while older clients query for an older version.

A COM component may not only run in the caller's context, it can also run in a different process or (through DCOM) on a different machine altogether. Inter-process communication and inter-machine communication are facilitated using client-side and server-side proxies, as explained in section 2.2.3 (referred to in COM as "proxies" and "stubs," respectively). COM supports both synchronous (blocking) and asynchronous (non-blocking) calls to components running out-of-process (Prosise 2000a).[7]

Marshalling may be handled automatically by COM, called *standard marshalling*. Advanced users that wish to handle all marshalling aspects themselves may elect to use *custom marshalling*. The latter may be preferable for performance-critical applications, as it makes it possible to handle certain operations without deferring to a remote server, thereby cutting down on inter-process or inter-machine calls. A custom-written client-side proxy could, for instance, cache data in the client process, and transparently operate on this state instead of consulting the remote object. Many of the benefits of custom marshalling may be reaped using *in-process handlers*, without the complexity of the former approach. An in-process handler

[7] To make inter-machine calls, DCOM uses an object-aware variant of the DCE wire format for remote procedure calls called Object RPC (Eddon and Eddon 1998). DCOM contacts a service known as the Service Control Manager on the remote machine to process and route inter-machine calls. This service performs a function similar to an ORB in CORBA, which is discussed in section 5.4 (Szyperski et al. 2002:341).

may elect to handle some operations locally, while delegating others to standard marshalling (Prosise 2000b).

COM can be used with the interface description language COM IDL, but as a binary standard, using this language is strictly speaking optional. COM IDL is an extended version of DCE's IDL, notably adding objects to the language (Hludzinski 1998). Microsoft's IDL compiler can generate client-side and server-side proxies, C/C++ language bindings, as well as *type libraries*. A type library is a non-textual, efficient representation of a set of IDL files, which may be deployed to end-users' systems as stand-alone files, or embedded as resources in shared libraries or executable files.[8] A type library is essentially a repository of type information available at runtime. The COM runtime system can read type libraries, and make the data therein available through the `ITypeInfo` interface. Language bindings are typically not generated directly from IDL files, but from type libraries, as type libraries are the entities that are deployed to end-users' systems.

Factories are used in COM to instantiate classes. For a class to be instantiable, there must be an implementation of `IClassFactory` available that can instantiate said class. A COM component that runs in-process (and thus is implemented as a shared library) must export a function that returns an object implementing `IClassFactory` for a given class UUID passed as an argument. A COM component that runs out-of-process on the same machine is implemented as a standard executable file, that when started registers its class factory with the COM runtime system (Goswell 1995).

One of the selling points of COM is that it enables what Microsoft calls *Automation*—the ability for one program, typically a script, to access and control another, which is often written in native code. A script written in Visual Basic can, for example, use the charting engine of Microsoft Excel through Automation. Automation allows applications to make their functionality available as a set of COM objects.

Automation implies that the validity of invocations are verified only at runtime, thus making use of very late binding (see section 4.2). The traditional solution in COM is to require that classes that wish to be accessible through very late binding implement the `IDispatch` interface, which is analogous to the `Scriptable` interface presented in Chapter 4. (Classes that are accessible using both late binding and very late binding, and thus implement `IDispatch` in addition to traditional, domain-specific interfaces, are said to use *dual interfaces*). `IDispatch::Invoke()` does not, unlike `Scriptable::InvokeOperation()`, take a string representing the name of the operation as an argument. Rather, it takes a dispatch identifier, which can be retrieved at runtime using `IDispatch::GetIDsOfNames()`, presumably for reasons of efficiency. If a component ships with a type library, the implementation of the `IDispatch` interface can be fully synthesized at runtime, or by simply forwarding calls to a system-provided implementation of `ITypeInfo`.

5.3 Delphi

In 1995, a few years after the introduction of Visual Basic, Borland introduced its first version of Delphi. At first glance, Delphi was very similar to Visual Basic. It came with an integrated development environment, complete with a visual designer used to construct user interfaces by dropping controls on forms, changing their properties and associating code with events.

[8] Type libraries can only represent a subset of the information contained in IDL files. A type library cannot represent multiple output arguments, for instance (Hunt and Scott 1999).

Delphi came with its own component model, but also supported VBX components built for Visual Basic. (Delphi 1.0 was a 16-bit environment. Later versions would add support for the 32-bit OCX/ActiveX components of later versions of Visual Basic.)

On closer inspection, the similarities were only skin deep. Delphi used the object-oriented Object Pascal language (later renamed the Delphi programming language), and came with a fast, optimizing compiler with a built-in assembler. Developers had full access to the Windows application programming interface from their applications, and could create shared libraries available to code written in languages such as C, C++ and even Visual Basic.

Contemporary versions of Delphi are owned and sold by Embarcadero Technologies, and target 32-bit Windows versions, with a planned 64-bit version. A separate product, using a slightly different programming language, is available for Microsoft's .NET platform as Delphi Prism.

Classes that descend from the `TComponent` class are referred to as "components" in Delphi, and may be manipulated in a visual editor. (This usage of the "component" word is inconsistent with the terminology used in this thesis.) Delphi has language-level support for properties that are used to visually customize such objects. Properties that are used in this way must come with extended runtime type information, partly so that their names can be presented to the developer in the visual environment. There is an additional "access specifier," `published`, for this purpose, which is identical to `public` in most respects, but stores extended type information. Delphi also has "controls" that feature graphical user interfaces at runtime (in other words, a slider or a check box is both a "control" and a "component," whereas a non-visual object that enables a developer to set up a database connection for use by data-aware controls is only a "component"). Such objects extend the `TControl` class which itself extends `TComponent`.

As of Delphi 3, code may be packaged as a special kind of shared library, a *package*, and these packages indeed qualify as components (with the notable exception that packages do not provide version information in a standardized way, and multiple versions cannot easily be loaded at the same time). A Delphi package may declaratively specify what other packages it depends on (Lischner 2000:7).

Delphi takes a very aggressive approach to supporting COM, and goes as far as making COM a part of the core language (although most of the COM support resides in the runtime system, and the COM language features are very much optional). The memory layout of Delphi classes is compatible with COM, enabling Delphi classes to function as COM classes if certain additional rules are followed (Lischner 2000:71). Interfaces are part of the Delphi language, at least partly for the sake of COM compatibility. Regardless of whether COM is used, all Delphi interfaces must extend the COM `IUnknown` interface, and be assigned a runtime name in the form of a UUID (the Delphi language provides a convenient UUID literal syntax for this purpose, making UUIDs very readable).

As all interfaces are known to be reference counted, Delphi calls `IUnknown::AddRef()` and `IUnknown::Release()` automatically; the former when references are assigned, the latter when references go out of scope (much like a C++ smart pointer). Delphi also has special language-level support for `IUnknown::QueryInterface()`: the `as` operator (which is normally used for "safe" typecasts[9]) uses the `IUnknown::QueryInterface()` operation when the first operand is an interface reference (Lischner 2000:54).

[9] A safe typecast checks the validity of the cast at runtime, using runtime type information, and throws an exception if the cast is not valid. Delphi also supports traditional Pascal typecasts, which are not checked.

While it can be argued that mandating the use of IUnknown as a base interface does not constitute bringing COM itself into the language (as interface navigation and reference counting are arguably useful for interfaces, regardless of whether COM is used), Delphi does have explicit language-level constructs that solely exist to support COM; a sampling follows:[10]

- Accessing an object through the IDispatch interface normally means that the compiler cannot perform static type checking, and that errors are only signaled at runtime. Delphi provides the dispinterface keyword for declaring such an interface, enabling errors to be caught at compile-time. dispid keywords can be used to manually assign the dispatch identifiers expected by IDispatch::Invoke() (Lischner 2000:179).

- Delphi introduces the safecall calling convention explicitly for the benefit of COM. This calling convention is identical to the stdcall calling convention expected by COM (and all other standard Windows functions), but adds exception "firewalls." As COM uses error return values in preference to exceptions, Delphi exceptions cannot cross a COM call boundary. The safecall calling convention solves this problem. If a Delphi method, declared with this keyword and serving as the implementation of a COM operation, throws an exception, the exception is caught and converted to the HRESULT return value expected by COM. Calling a COM operation works in much the same way—the HRESULT value is checked automatically, and an exception is thrown if the return value does not indicate success (Lischner 2000:326).

- In order to refer to a variable whose type is not known, a strongly typed language needs what is sometimes called a *variant type*. Variables of such a compile-time type can change their type at runtime. The ScriptableReturnValue_t type, shown on page 63, is an example of a simple variant type. Variant types are often, as in the preceding listing, implemented as a tagged union in C. Delphi has a dedicated type for variants, simply named Variant. In particular, Delphi also has the type OleVariant, which is restricted to COM-compatible types (Lischner 2000:270). Both are built-in primitive types, and have no formal declaration outside of the compiler implementation.

Delphi's language-level support for COM makes for unusually clean COM code, although it is unusual with language features specifically tailored to a particular platform.[11] Most of Delphi's COM support is confined to its runtime system, though, which provides classes such as TComObject that can be extended to easily create classes compatible with COM, and TComObjectFactory which implements IClassFactory (Calvert 1999:386). The integrated development environment includes COM "wizards" that help developers with various COM-related tasks without requiring much knowledge of COM's inner workings. For instance,

[10] As COM has no native support for implementation inheritance, forwarding calls is often more natural than using implementation inheritance when writing a COM class (and arguably better practice in other scenarios as well, given the reservations expressed on page 70). The Delphi keyword implements is, while not COM-specific, very useful when used with COM, as it introduces language-level support for forwarding. Using this keyword, an object, implementing a set of interfaces, can delegate the implementation of any number of its implemented interfaces to other objects it maintains references to. While this can obviously be implemented by manually forwarding calls to the internally referenced objects, this keyword makes for less verbose code.

[11] Languages such as C# and Java allow developers to associate code with arbitrary metadata. Had Delphi had such support at the time when COM support was introduced, it is conceivable that some of its COM-specific language support would have been in the form of such metadata. This is the route taken by C#/.NET for COM compatibility (discussed on page 94).

Delphi can automatically wrap a Delphi control as an ActiveX control, usable in a variety of environments, including Visual Basic. Delphi language bindings can be generated for COM components by “importing” their type libraries.

Delphi is a good example of a language that integrates very closely with COM, making writing and using COM objects less daunting. Microsoft also provides such tools, notably in the form of Visual Basic 6, and the Active Template Library, a set of template-based C++ classes for creating COM objects.

5.4 CORBA

The Common Object Request Broker Architecture (CORBA) is a standard of the Object Management Group (OMG), one of the largest software consortia in the computer industry. It enables objects running on different platforms, on different machines and written in different programming languages to interoperate. The OMG does not create software per se; it creates specifications that are implemented by different vendors. There are a large number of CORBA implementations—some commercial, others open source—that to a fairly large degree can be substituted for one another (unless proprietary extensions are used). The primary focus of CORBA is on enterprise systems, and as such, CORBA standardizes a large number of services that are useful for such systems, handling aspects such as database transactions, concurrency, object persistence, licensing, event notification and security (Szyperski et al. 2002:240). CORBA’s naming service is especially important, as it allows clients to look up server objects by name (Aleksy et al. 2005:269). Despite CORBA’s early focus on the enterprise, there are also specifications that enable CORBA implementations on resource-constrained embedded systems (Schmidt and Vinoski 2001). CORBA’s strengths aside, there has been some concern lately that interest in CORBA is waning (Henning 2006).

The CORBA runtime system consists of an Object Request Broker (ORB), which routes calls from one object to another and returns potential return values. An ORB handles all invocation specifics, such as finding the target object and marshalling arguments. All object invocations, regardless of whether the receiving object runs in-process or out-of-process, go through the ORB. By mandating the use of runtime software for all object invocations, CORBA can effectively hide differences between communicating objects—what languages they are written in, on what machines they are running and the operating system used—without requiring that their interfaces all look the same in memory. Thus, CORBA is, unlike COM, not a binary standard.

All objects are accessed using interfaces, which are specified in a CORBA-specific IDL dialect. It is this language, in conjunction with formalized language bindings, that serves as the standardization mechanism of CORBA, not the memory representation of interfaces. The IDL dialect is incompatible with that used by COM, and provides a few features unsupported by COM IDL, such as support for exceptions, modules and multiple interface inheritance. Aside from an ORB, every CORBA implementation also ships with an IDL compiler. One of the primary uses of a CORBA IDL compiler is to generate language-specific client-side and server-side proxies (known as *stubs* and *skeletons*, respectively), which are used to issue and deliver object requests. Stubs are used by clients that can check invocations at compile-time, such as those written in C or C++.

Clients that invoke operations without having compile-time knowledge of them, and thus use very late binding, cannot use a stub. Rather, they must use the Dynamic Invocation

Interface (DII) of CORBA. As this interface is provided by the ORB, the target object remains oblivious to the invocation mechanism used to communicate with it (this is in stark contrast to COM, which requires objects reachable through very late binding to implement the `IDispatch` interface themselves). Dynamic invocations can be both synchronous and asynchronous. CORBA also supports the Dynamic Skeleton Interface (DSI), which allows objects to receive requests without having compile-time knowledge of the interfaces they implement.[12] The runtime type information required by these interfaces is stored in CORBA's Interface Repository, which is analogous to a set of COM type libraries. Like type libraries, the Interface Repository can be traversed programmatically (Vinoski 1997).

CORBA defines a number of standardized language bindings, which make it possible to access and implement CORBA objects in a given language.[13] (As a binary standard, COM does not have to define any language bindings—it simply defines a binary standard for interfaces, and expects compilers and script hosts to adapt to this standard.) Language bindings are implemented by an *object adapter*, leaving the ORB to focus on language-agnostic parts of the runtime system. An object is an abstract entity that clients refer to through *object references*, which as of CORBA 2.0 have been standardized as Interoperable Object References, or IORs. An IOR contains all the information necessary to contact the object, including the IP address and port number for a remote object. The implementation of an object is known as a *servant* (Aleksy et al. 2005:15). The object adapter is responsible for binding an object reference to a concrete servant, which can serve the request made on the object.

While CORBA does not define a binary standard for interfaces, it does standardize the wire format used over a network. Prior to CORBA 2.0, there was no standardized wire format. As a result, vendors used their own proprietary formats, meaning that two objects using ORBs from different vendors could not communicate. As of CORBA 2.0, all implementations must support the standardized Internet Inter-ORB Protocol (IIOP), which is an implementation of the General Inter-ORB Protocol (GIOP).

CORBA 3.0 introduces the CORBA Component Model (CCM), which formally defines components in the CORBA context. CCM defines an execution environment for hosting components in the form of an application server, thus removing the need for developers to build ad-hoc server solutions. In addition, CCM introduces standards for packaging, assembling, and deploying components, enabling CORBA implementations to activate components remotely (prior to CCM, there were no such standards, making users rely on non-standard solutions). CCM augments the standard IDL language with the Component IDL (CIDL) language, which enables developers to describe components and their interfaces (Schmidt and Vinoski 2004).

5.4.1 Implications of not using a binary standard

The CORBA approach of standardizing on an IDL dialect and on formalized language bindings, instead of on the memory layout of interfaces as COM does, has a few implications. First, in-process object invocations are by necessity slower than had a binary standard been used. A CORBA object invocation is normally done indirectly through an ORB, which marshals all arguments before contacting the target object. In-process COM calls are identical to calling a C++ virtual member function, and thus entail no such performance penalty.

[12] This is useful for solutions that act as bridges between CORBA and other component models.

[13] CORBA language bindings are formally called "language mappings."

Some ORBs use the same stubs and skeletons for in-process calls as for remote calls, which means that in-process calls are marshalled, sent over the network loopback interface, and unmarshalled by the ORB before finally reaching the servant. The poor performance characteristics of this strategy can be mitigated using so-called *collocation optimizations*. Schmidt et al. (1999) describe two such optimization strategies implemented for the open-source TAO ORB, one of which routes calls through the object adapter, and one which forwards calls directly from the stub to the servant. The first strategy improves performance by several orders of magnitude, but is still twice as slow as a call through a dispatch table. The second strategy almost closes the performance gap, but is no longer CORBA-compliant (cutting the ORB out of the loop means that it can no longer perform any of its services, such as intercepting calls to enforce a threading or security policy). Also, it only works if the client and servant are known to be binary compatible, that is, if they have been written in the same language, or target the same binary object model. Despite primarily being designed for distributed systems, this research demonstrates that CORBA's performance can be quite competitive with that of a binary standard.

A second implication of CORBA's design is that making a solution compatible with CORBA is likely less onerous that making it compatible with COM, as a COM-compatible language must comply with the binary standard of COM, if native objects of that language are to be regarded as COM objects.[14] CORBA's requirement that a runtime layer, in the form of an ORB, is used may entail a slight performance penalty, but it does free implementations from having to adopt a binary standard. Enabling a scripting language to be used for writing COM classes, for example, is possible, but requires script hosts to use internal data structures compatible with COM's binary standard, or alternatively synthesize such structures at runtime. CORBA's requirement that an ORB must be used to invoke operations successfully decouples the implementation structures used for objects from the ability to invoke operations on such objects. COM's approach is effective from a performance point of view, though, and well-suited to environments where classes are expected to be written in a compiled object-oriented language such as C++, and accessed from a scripting language, such as older versions of Visual Basic.

A third implication of objects always communicating through an intermediary is that calls to objects can be trivially intercepted, which is more difficult if no such intermediary is present at all times. Some component models allow users to request the use of services in a declarative manner, which is realized by intercepting calls. This is the subject of Chapter 7.

5.5 Java

The first few years of what would eventually be known as Java were difficult. Sun Microsystems, the owner of the technology, initially struggled to find a market for its technology. When the company failed to find a niche in TV set-top boxes in the middle of the 1990s, it turned to the emergent World Wide Web. A deal was struck with Netscape, a maker of web browsers, to bundle the Java technology with their products. Through small programs running within

[14] A programming language with no native objects, or native objects that do not adhere to COM's binary standard, may still make COM objects available if they, like C, support structures, pointers and calling functions through pointers. This approach can be used with a C++ compiler whose native objects are incompatible with COM, for instance.

the confines of a web browser, known as *applets*, Java enabled interactivity and animation on what was then a largely static Web (Bank 1995; Byous 1998).

Today, Java enjoys considerable success in the enterprise space with its Enterprise Edition. This edition provides a component model known by the name of its components—Enterprise JavaBeans (EJBs).[15] EJBs give access to many of the enterprise services enumerated in section 1.7, such as declarative handling of database transactions. The Micro Edition of Java is used in the embedded space, and is commonly run on cellular phones, powering many of the downloadable applications and games on Java-compatible phones. A variety of solutions exist for running Java code, partly or fully, directly on embedded hardware platforms where it may not be practical to use just-in-time compilation (Libby and Kent 2009). The Standard Edition runs on desktop machines, and provides a large number of standard libraries, including ones that allow programmers to create graphical user interfaces. Java applets are still used by modern web sites, but perhaps not to the extent originally envisioned. Sun released most of the Java code under an open-source license in 2007.

Java is an object-oriented language supporting exceptions and parameterized types *(generics).* Java programs are typically compiled to machine code (bytecode) for a stack-based virtual machine, known as a Java Virtual Machine (JVM). The virtual machine implementation may interpret all bytecode, but most contemporary implementations compile at least some of the bytecode to native code to speed execution (using just-in-time compilation).[16] This has the benefit of enabling Java programs to run on any platform for which a JVM implementation exists. Java provides a strong security infrastructure, which ensures that untrusted code, such as applets, can be safely allowed to run.

5.5.1 Repartitioning the platform

Component models such as COM provide a wealth of functionality, most of which is not related to software components per se. A memory management strategy that can cope with independently written parties, an object model and access to type information at runtime are enablers of component technology, but need not be considered part of it.

The designers of Java had the luxury of designing a new platform from the ground up, complete with a new instruction set, and were thus not hamstrung by the requirement that traditional compilers, targeting traditional processors, be used. The partitioning of functionality thus looks quite different in Java: much of the technology often associated with component technology has completely migrated to the core platform.

Object model

COM provides both a component model and an object model. The Java language has an object model built-in. In recognition of the problems caused by allowing multiple implementation inheritance (Szyperski et al. 2002:111), Java only supports single implementation inheritance, but does support multiple interface inheritance. Classes and interfaces are contained in *packages*, which both serve as Java's namespace mechanism and as a means of access control. Packages

[15] Java also comes with a technology named *JavaBeans*, which is unrelated to Enterprise JavaBeans. Such JavaBeans are at their core ordinary Java objects that follow certain conventions, making it possible to customize their properties in visual tools. JavaBeans are touted as components, but are not compatible with the view of software components adopted by this thesis.

[16] Third-party solutions exist that enable ahead-of-time compilation, such as GNU's GCJ.

are identified using the same strategy as that presented in Chapter 4, that is, top-level Internet domain names are used as part of the name to ensure its uniqueness, forming names such as `org.organization.project`. Compile-time names of classes and interfaces are qualified by the names of their packages, forming names such as `org.organization.project.SomeInterface`. Packages contain classes and interfaces, which may be declared as being either public (and thus accessible to all classes) or private to their parent package.

Not only does the Java language provide classes and objects, the instruction set of the JVM is also object-savvy.[17] All languages targeting the Java instruction set can use classes defined in a different language targeting the same (virtual) machine, as there is nothing language-specific about the definition of a class or an interface. Hence, Java standardizes objects on the level of its instruction set, COM on the memory representation of interfaces and CORBA on an interface description language coupled with formalized language bindings.

Reflection

Java makes type information available at runtime through reflection, allowing a program to, for instance, iterate over all methods provided by a certain class and invoke one based on whether its name contains a given substring. (This can be used by frameworks that rely on naming conventions in lieu of more traditional mechanisms, such as having classes implement certain interfaces.)

As a result of this support, a Java program can inspect an interface at runtime and synthesize a class implementing it, which is especially useful for creating proxies at runtime. Having support for reflection as a core part of the platform also means that supporting very late binding is close to trivial. A script interpreter written in Java can easily expose a function library to the scripts it executes, or even give access to the complete Java class library.

Memory management

Java uses automatic garbage collection in preference to manually managing memory, thus eliminating many of the problems associated with reference counting (cyclic references and programmers forgetting to add or remove references).

Error handling

Java uses exceptions to signal errors. Unusually, Java introduces the notions of *checked* and *unchecked* exceptions. Checked exceptions must be either caught and handled, or the method throwing the exception must explicitly list the exception (or one of its ancestor classes) in its *throws* clause. Unchecked exceptions behave as exceptions in other languages, that is, they automatically propagate if not explicitly handled. At best, checked exceptions make the programmer aware of potential error conditions that should be handled (such as opening a file that may not exist), at worst, it introduces verbose code to handle exceptions that are *a priori* known never to be thrown (such as compiling a regular expression that is stored as a string literal). Whether an exception is checked or unchecked is determined statically, by checking to see if the exception class is in an *is-a* relationship with a system ancestor class.

[17] Had, say, the ubiquitous *x86* instruction set provided object-savvy instructions (such as one for invoking a virtual operation using late binding), there would be little disagreement over how to implement this functionality in compilers for two different object-oriented languages. (That is not to say that having support for object-orientation at the silicon or microcode level would be appropriate from a cost-benefit perspective, though.)

Distributed computing

Java has support for distributed computing in the form of Remote Method Invocations (RMI). The original version of RMI only supported communication between two Java virtual machines, using a custom wire format; current RMI versions also support CORBA, and can thus also use the CORBA IIOP wire format.[18] This version is called RMI-IIOP. RMI is used internally by other parts of Java, such as Enterprise JavaBeans.

Objects that are to be available remotely must explicitly implement an interface to that effect—`java.rmi.Remote`. RMI provides a registry that such objects need to register with. Once a reference to a remote object has been acquired, methods can be called on the remote object as though they were local—the RMI runtime system handles marshalling. Objects that implement the `Remote` interface can be passed by reference. RMI passes objects by value that do not implement this interface, but do implement `java.io.Serializable` (which is an empty "marker interface" that signals that the Java runtime system is permitted to convert objects implementing it into byte streams). RMI transmits the byte stream over the network, and deserializes the object on the remote server. One distinguishing feature of RMI is that if the JVM running on the remote server does not have a local copy of the class that the passed object is an instance of, RMI can dynamically transmit the class over the network, which is then loaded into the server JVM. This makes it possible to offload expensive operations to a remote server without requiring that the server has compile-time knowledge of the tasks it is to execute.

The first version of RMI required the compile-time presence of both client-side and server-side proxies (called *stubs* and *skeletons*, respectively). Developers were expected to generate proxy implementations using a tool shipped with the Java distribution—`rmic`, analogous to a COM or CORBA IDL compiler. Whereas an IDL compiler operates on IDL files, `rmic` operates on files containing compiled Java bytecode. As the Java language, as well as the instruction set of the JVM, already have an interface concept, there is no need for a specialized interface description language.

As of version 1.2 of Java, server-side proxies are no longer needed. A generic server-side proxy is part of the platform, and uses reflection to reflectively invoke the methods of the remote object. Also, as of version 5.0, client-side proxies no longer need to be generated manually—they are synthesized dynamically (that is, created at runtime), again using reflection. (Note that a client always accesses a remote object through an interface. Thus, even if the actual client-side proxy implementation is synthesized at runtime, the client still has compile-time knowledge of the methods that the remote object makes available, enabling compile-time type checking.)

RMI exemplifies that distributed computing does not need to be complicated. RMI requires no interface description language, as the platform is already interface-savvy, and thanks to the pervasive use of runtime type information by the Java platform, it can do without generated client-side and server-side proxies. This is vastly simpler than the solutions used by COM and CORBA.

5.5.2 Modularity woes

JAR files are the preferred deployment format for Java libraries. Such files contain compiled Java bytecode, as well as a "manifest" file holding metadata. JAR files cannot be considered

[18] In fact, there is a formal Java language binding for CORBA, and Sun's implementation of Java even comes with a lightweight ORB, an IDL compiler and naming service (collectively, Java's CORBA services are known as "JavaIDL").

true components, as they have no identity at runtime. All packages that are part of a JAR file are placed in the same global namespace as packages loaded from a different JAR file.[19] This has subtle implications that serve to illustrate the necessity for components to be identifiable at runtime.

For simple cases, JAR files work well. Java's use of packages ensures that two classes or interfaces that share the same name and are provided by two different libraries can co-exist, as they are placed in different packages. The system breaks down, however, when an attempt to load multiple versions of the same JAR file is attempted.

Complex Java projects often have a large number of dependencies, which can number in the hundreds. A dependency often has dependencies of its own, resulting in complex dependency graphs. One JAR file may be dependent on a certain version of a library, while another JAR file is dependent on another version. Java simply attempts to load the first version it encounters,[20] which may or may not be compatible with the JAR file expecting a different version. Compounding the problem, if the loaded JAR file does not contain a class or interface that is part of a JAR file that was initially not loaded, Java will attempt to load the desired class or interface from the latter JAR file. The end result is a toxic mix of incompatible classes and interfaces, resulting in runtime errors, or worse, unpredictable behavior.

Java provides access specifiers that allow developers to control access to classes, interfaces, and fields. A class or interface can either be exposed to all other classes running in the same virtual machine using the `public` access specifier, or it can be designated *package-private*, meaning that a particular type may not be accessed from outside its parent package. It is not, however, possible to expose a class or interface only to the JAR file in which it resides, which illustrates another problem with JAR files not having an identity at runtime.[21]

5.5.3 True Java components

While JAR files cannot in themselves be considered true components, Java does ship with a proper, if domain-specific, component model in its Enterprise Edition. Enterprise JavaBeans (EJBs) are hosted by an application server and have access to enterprise services, often by setting declarative attributes. An EJB entity bean can, for instance, declaratively request that the application server should handle its database access by transparently storing and loading its state. (Entity beans represent persistent data typically stored in a relational database (Mukhar et al. 2005).)

EJBs cannot interfere with each other in the manner described in the previous section as their containers do have runtime identities. EJBs achieve this using custom *class loaders*. Class loaders are Java classes that descend from `java.lang.ClassLoader` and are responsible for loading classes and interfaces from compiled bytecode. A typical class loader loads classes

[19] Packages that house the Java class library and packages that are part of user-installed extensions are actually part of different namespaces.

[20] This issue is related to the mechanism used by Java to load classes into a virtual machine, known as *class loaders* (discussed on the current page). The application class loader, which is normally responsible for loading classes that are neither part of the core runtime system nor part of a system extension, consults the *classpath* to find classes. The classpath is a delimiter-separated string containing the locations used to find classes, either directory names or the locations of JAR files. The application class loader is satisfied when it encounters a directory or JAR file that contains the sought class or interface. Once a class or interface has been found, no further classpath entries are consulted.

[21] The common work-around is to define a package named "internal," and place classes and interfaces that should not be exposed to the outside world in that package or its subpackages. This informal means of limiting access to internal types cannot easily be enforced at compile-time, though.

from the file system, but custom class loader can also load classes from, say, in-memory data structures or over a network. Internally, a Java virtual machine represents a class or interface using not only its given compile-time name, but also using the class loader responsible for loading it (Lindholm and Yellin 1999). A Java runtime name thus consists of a pair: its compile-time name and its defining class loader. This allows a virtual machine to differentiate between two objects that share the same class and package names but were loaded using two different class loaders. A class loader is expected to delegate calls to a parent class loader before it tries to load a class itself. Class loaders thus often form an implicit tree. At the root of this tree is the "bootstrap" class loader, which loads classes from Java's core runtime system. The bootstrap class loader is thus always responsible for loading the definitions of core classes such as `java.lang.Object` (from which all classes ultimately descend).

Class loaders are a very useful mechanism on which to build true Java components, and as class loaders are normally implemented entirely in Java, there is no need to resort to native code. Corwin et al. (2003) present such a system, MJ, which attempts to solve the two fundamental problems discussed on page 88. MJ components must list their dependencies and what packages they provide as part of their metadata. Using custom class loaders, MJ ensures that dependent types are available (by simply calling on the services of the class loader associated with the dependent component), and that types that are not explicitly exported from a given component are not made available to the outside world. (When MJ is used, the class loaders in the system no longer form a tree; they form a directed acyclic graph, as class loaders can delegate calls to an arbitrary number of other class loaders, representing components which the current component is dependent on.)

In industry, the ideas of MJ are best embodied by a popular Java component model named OSGi. OSGi components are called "bundles," and are deployed as ordinary JAR files, with the sole difference being that the manifest file holds additional metadata similar to that mandated by MJ, such as lists of dependencies and of packages that are exported from the bundle. As an OSGi bundle is a standard JAR file, it can be loaded both by a specialized OSGi runtime system (enabling it to be used as a true component), and in the traditional fashion. OSGi defines a rigid versioning scheme, allowing a bundle to require a specific version of another bundle, or declaring that it can accept a range of acceptable versions.[22]

OSGi bills itself as the "dynamic module system for Java," meaning that bundles can be loaded and unloaded at runtime. This is important to application servers, which should suffer as little downtime as possible. The dynamic nature of OSGi makes it possible to install a new version of a bundle without taking down the server, and without affecting other bundles. To achieve this, OSGi defines a strict life cycle for bundles.

In contrast to other component models studied, OSGi only attempts to enable software components that run in-process (that is, bundles that all run in the same virtual machine). A distributed version is reportedly in the works (Taft 2008).

OSGi bundles adhere perfectly to the definitions of Chapter 1. Bundles are protected from one another, enabling multiple versions to co-exist, and declare their dependencies and what they themselves provide declaratively. OSGi is cleanly layered on top of the Java platform, and as such, OSGi bundles are usable in binary form on any platform for which a Java implementation exists. A handful of commercial and open source implementations are

[22] The recognized OSGi "best practice" is not to request that a specific, named bundle be present, but to import a particular version of a package, making it possible to move packages from one bundle to another without unduly disturbing dependent bundles (Bartlett 2009:60).

available. As of the middle of 2009, OSGi is making its way through the Java Community Process, and may become a standard part of a future version of the Standard Edition of Java.

5.6 .NET

In August 1998, Microsoft shipped version 6.0 of their Java development tool, Visual J++. This product turned out to be controversial, partly because it allowed developers to very easily call on the services of the Windows platform, thus violating what Sun saw as one of the key selling points of the Java platform: its platform-agnosticism. Visual J++ 6.0 was billed as having "the ease of Java, the power of Windows." Microsoft saw in Java an innovative new language, and wanted to leverage the Java language to enable developers to more easily build Windows applications. A lawsuit was filed by Sun, and Microsoft was eventually forced to stop developing their Java product (Microsoft 2005; Lohr 1998). In response to these events, Microsoft developed their own in-house platform .NET, which is reminiscent of the Java platform, and a new flagship language for .NET, named C#, that has many traits in common with the Java language.

Anders Hejlsberg, the former architect of Delphi, was the architect behind Visual J++ 6.0. He went on to become architect of the C# programming language, and a key participant in the development of the .NET platform. He shared his thoughts on the Java experience in an interview (Microsoft 2005):

> [The Java experience] gave us clarity [...] around creating .NET [...], taking control of our own destiny, and building a platform where we could truly innovate and where we could build the stuff that we needed to build for our customers as opposed to trying to shape a competitor's platform, which effectively is what we were doing. [...] The development environment we had on Windows at the time [...] had many names, COM, OLE, ActiveX, DNA [...]. We kept rebranding the same bucket of bits. [...] It was [...] a very low-level experience, there were all sorts of... Registry and GUIDs and HRESULTs and interfaces and stuff you had to deal with. Horribly complicated. [...] Simplicity was getting lost. [...] Developers were voting with their feet [and abandoning the Windows platform for the Java platform], so we were kind of in trouble. [...] There was a lot of handwringing about where do we go. I think there were two camps, the evolutionaries and the revolutionaries. The evolutionaries said "we got to fix COM." [...] It seemed to me a pretty big fix. And then there were the revolutionaries, and I was pretty firmly in that camp, that said "we got to build a new development experience, we got to clean up all this stuff—of course we need to interoperate with it, but we got to have a fresh start."

5.6.1 Technical foundation

The .NET platform is in many respects similar to the Java platform. The core of .NET is a stack-based virtual machine called the Common Language Runtime (CLR), which executes object-savvy bytecode.[23] The CLR typically compiles bytecode to native code using a just-in-time compiler, but there are also provisions for ahead-of-time compilation, which is often used with embedded systems. Microsoft refers to bytecode as "managed code," and to machine code

[23] The instruction set of .NET is formally divided into the "Base Instruction Set" and the "Object Model."

that runs natively on a CPU as "unmanaged code." To write low-level managed code, one can use the Common Intermediate Language (CIL), which is an assembly language that targets the CLR. Like the JVM, the CLR manages memory automatically through garbage collection, and natively supports exceptions (although the CLR does not have Java's distinction between checked and unchecked exceptions; all are unchecked).

.NET has been standardized as the Common Language Infrastructure (CLI), which has been implemented by vendors other than Microsoft. Notably, Novell has created the Mono implementation, which is available for a wider range of platforms than .NET, such as Linux, various Unix platforms and Apple's Mac OS X. .NET has a counterpart to the Micro Edition of Java in the .NET Compact Framework, which is a version of the .NET platform designed to run on embedded systems, such as cellular phones. Libby and Kent (2009) have done research on implementing the CLI in hardware for the benefit of embedded systems.

While it is possible for languages other than Java to target the JVM (and many do), .NET was specifically created to allow programs written in different programming languages to run on the same virtual machine. By contrast, the JVM was specifically designed for programs written in the Java programming language. This can be seen in that while both the JVM and the CLR have explicit support for classes, only the CLR allows methods outside of classes, thus enabling procedural languages to easily target the CLR.[24] Also, a number of specifications are associated with .NET that enable different languages to interoperate. The Common Type Specification (CTS) formally defines the type system used by various .NET languages. The Common Language Specification (CLS) is a subset of the CTS, and is a set of rules that all languages targeting the CLI should comply with in order to be interoperable with other CLS-compliant languages (Troelsen 2007).

Visual Basic and C# are Microsoft's two flagship languages targeting the CLR. Visual Basic has been retrofitted as a pure .NET language, and does not retain source-level compatibility with its previous incarnations. C# is in many respects similar to Java, and is an object-oriented programming language supporting interfaces, single implementation inheritance and parameterized types (generics). C# does sport some noteworthy features that are not in Java, though, such as language-level support for properties and lambda expressions. Language-integrated queries (LINQ) can be used to query databases (instead of including SQL string literals that are parsed at runtime, as is done in Java and most other languages).

CLR has rich support for runtime metadata, which is accessed programmatically through reflection. Unlike Java, the type system of the CLR (the CTS) is aware of generics, and makes this information available through reflection.[25] .NET Remoting, a .NET technology for realizing distributed computing and similar to Java's RMI, is one of many technologies to take advantage of reflection. By using metadata available at runtime, .NET Remoting is, like RMI, able to use dynamically synthesized client-side proxies and generic server-side proxies instead of requiring users to generate them statically (Rammer and Szpuszta 2005).

[24] A procedural language can target the JVM by simply implementing procedures as static methods that are part of an unnamed class.

[25] Generics are also available in Java, but in order to preserve backwards compatibility, Sun decided to make them strictly a compile-time tool, and as such, generic type information is not available at runtime. (Java removes generic type information through a process called "type erasure").

5.6.2 Interoperating with native code

When Microsoft designed Visual J++ 6.0, they chose not to implement the Java Native Interface (JNI), which was (and remains to this day) the means of accessing native code from Java (and vice versa). Instead, Microsoft opted to implement their own Windows-friendly solution.

In Java, any method may be declared with the `native` keyword. Such a method is expected to be implemented in native code, and is part of a shared library that runs in the address space of the JVM (and can thus easily crash the JVM, or corrupt its state). JNI mandates that all native functions follow certain naming conventions, and take JNI-specific arguments. The first such argument, of type `JNIEnv`, gives access to the services of the JVM using a dispatch table (the `JNIEnv` argument happens to use the same binary memory layout as COM and most C++ compilers; this solution enables native code to be insulated from the inner workings of the JVM). Native code must interact with the JVM in order to process an invocation. This includes converting strings to a format usable by native code, throwing exceptions and converting local object references to global object references (that the native code can store for later use, without fear of the garbage collector prematurely collecting them). To use the services of a native library, a Java developer must manually write a native wrapper for it that forwards all calls to the native library.

Creating such wrappers is a burdensome task, especially if many native libraries are used. As Microsoft intended for their version of Java to be a better way of writing Windows applications, they needed a solution that would make calling on native platform services almost effortless. Besides introducing their own JNI-like technology (termed the Raw Native Interface), Microsoft also introduced a technology known as J/Direct to enable Java developers on Windows to easily make use of Windows services (Eckel 1998).

Native methods that were to use J/Direct were declared with the `native` keyword. In contrast to JNI, however, J/Direct methods were preceded by a Java comment with content that was taken into account by Microsoft's Java compiler. At a bare minimum, such a comment had to specify the name of the shared library that was to be loaded. J/Direct then took care of marshalling all arguments into the format expected by the native code, which was not privy to the fact that it was called through J/Direct. Hence, native libraries could be used directly, and no native wrappers had to be written. Special J/Direct directives could be used to handle callbacks, and to instruct the JVM to marshal strings as wide strings or as strings using single-byte character sets. Microsoft even provided a ready-made Java package, containing the Java equivalent to the function prototypes and constants found in the C header files shipped as part of the Windows software development kit, enabling developers to effortlessly use Windows services.

Visual J++ 6.0 also featured deep COM integration, enabling Java classes to seamlessly access COM servers and conversely make Java classes available to native code through COM. To enable Java code to access COM servers, Microsoft shipped a tool that generated Java bindings from a type library. The COM objects used from Java were proxies partly implemented in native code, that marshalled arguments and forwarded calls to the native COM object. `HRESULT` return values that indicated failure were transparently converted to Java exceptions.

Java classes were COM-enabled using a tool that directly parsed compiled Java files and generated a type library, all without involving an interface description language. The generated COM interfaces could optionally be dual interfaces, enabling both late and very late binding (thus making it possible to use the Java COM objects from both native code and interpreted code). Microsoft's JVM provided COM clients with COM-compatible proxy objects that marshalled arguments and forwarded calls to Java code.

The Platform Invocation Services of .NET, commonly referred to simply as P/Invoke, are broadly similar to J/Direct. P/Invoke declarations are not contained in comments, but in .NET attributes, which are a formalized means of associating metadata with language constructs (Java "annotations" are roughly equivalent). Microsoft does not ship a .NET equivalent to the C header files that enable access to Windows functions, meaning that users must manually author the P/Invoke metadata (Bukovics 2006).

.NET also features deep COM integration, much like that provided by Visual J++ 6.0. Making COM objects available to .NET involves generating bindings from type libraries. To call on the services of a COM object, .NET uses a proxy called a Runtime Callable Wrapper, which handles marshalling and forwarding. Conversely, .NET can expose managed code through a COM Callable Wrapper, which, again, is a proxy. In its simplest form, .NET exposes the complete class interfaces of classes (including methods inherited from the base class, `System.Object`). It is recommended that interfaces specifically meant to be exposed through COM are written instead. Again like Visual J++ 6.0, .NET supports both late and very late binding through COM dual interfaces.

The first versions of .NET enabled the use of enterprise services solely through COM+. The COM+ support is integrated directly in .NET, and as such, developers do not need to use COM+ services through COM interfaces. As of this writing, some of the COM+ services are being replaced with pure .NET services, such as those provided by the Windows Communication Foundation.

5.6.3 True .NET components

In .NET, all managed code is contained in *assemblies*, which are independent, versioned entities that explicitly declare their dependencies. A .NET executable file is an assembly with an entry point, whereas a .NET shared library is an assembly without such an entry point. An assembly may consist of one or many *modules*, each of which can be written in separate languages targeting the CLR. Assemblies can be private or shared; shared assemblies are placed in the Global Assembly Cache for use by other assemblies (Troelsen 2007:347).

Assemblies typically contain only managed code, but may also be "mixed," meaning that they contain both managed and unmanaged code. Such assemblies can be produced by Microsoft's C++ compiler. A C++ class may be written using both managed and unmanaged code, making it particularly easy for managed code to interoperate with unmanaged code, without the use of P/Invoke or the COM interoperability services.

Unlike an OSGi bundle, there is no metadata on the level of an assembly that declares which types should be visible to other assemblies. .NET does provide such metadata, but as normal access specifiers. In C#, types can either be declared as `internal` or `public`. Internal types, which is the default access specifier if none is given explicitly, are not visible outside of the defining assembly.

An assembly cannot, unlike an OSGi bundle, be loaded and unloaded dynamically. However, assemblies are contained in so-called "app domains," which do have this feature. If desired, an app domain can house only one assembly, in effect making it possible to load and unload singular assemblies.

With assemblies, .NET features true components as part of the core platform.

CHAPTER 6

The (Sony) Ericsson way

In early 2001, Sony and Ericsson announced plans to merge their respective cellular phone businesses. Sony and Ericsson had both struggled in the marketplace, and it was felt that combining Ericsson's expertise in telecommunications with Sony's experience in consumer electronics would create a stronger business than either Sony or Ericsson could manage on its own. The joint venture, equally owned by Sony and Ericsson and named Sony Ericsson, began operations on October 1, 2001.

The hardware and software platform used in Ericsson's phones was chosen as the base on which to build consumer phones. As a result, Ericsson created a subsidiary, Ericsson Mobile Platforms, to license its cellular phone technology to a range of companies, including Sony Ericsson (Kornby 2005). As of February 2009, this business is part of a company named ST-Ericsson.

When Ericsson's cellular phone business became Ericsson Mobile Platforms and Sony Ericsson, the cellular phone software had to be split into two halves, one focusing on core services and the other on user-facing applications and services higher up in the system. Ericsson kept the parts essential to offering a complete cellular phone platform while Sony Ericsson assimilated the rest.

Separating these two halves necessitated the creation of a formalized barrier between the two, severing the direct ties between user-facing applications and the services offered by the cellular phone platform. To facilitate this, an object model was created as part of the new Ericsson Component Model (ECM). This object model came with support for dynamic dispatch, enabling freestanding interfaces and thus the separation of interface and implementation.

The formalized barrier was built using ECM, and was named the Open Platform API (OPA). To this day, OPA is used to access platform functionality, and is organized in a number of categories, each representing an aspect of the platform. Through OPA, developers interact with the ST-Ericsson platform, enabling activities such as initiating phone calls, sending text messages, playing MP3 audio files and rendering 3D graphics. OPA also serves as the interface to the underlying real-time operating system.

6.1 The Ericsson Component Model

The object model of ECM is based on that of COM (Ghosh et al. 2005). ECM defines instantiable classes, which can implement an arbitrary number of interfaces, and when instantiated form objects.[1] The memory layout of interfaces is almost identical to that used by COM—the binary standard of the ECM object model is thus concerned with the in-memory representation of interfaces, which use dispatch tables to realize late binding. Figure 6.1 depicts the memory layout of ECM interfaces. Very late binding is not supported, and only minimal runtime type information is kept (enabling interface navigation).

Interfaces and classes are assigned runtime names in the form of UUIDs. Interfaces may extend one other interface, with the exception of `IRoot`, which is the root of the interface hierarchy.[2] `IRoot` is identical to COM's `IUnknown`, and ECM thus uses reference counting to manage the lifetime of objects and supports interface navigation. Errors are signaled through return values—the equivalent to COM's `HRESULT` return type is the `RVoid` type, which contains 16 bits of error information divided into a number of fields. The remaining 16 bits may be used to transmit actual return values in addition to the error information—the `RSel` type is used to return integer values and the `RBool` type is used to return boolean values. Return values are typically transmitted using output arguments, though.

ECM uses factories to instantiate objects, which are always allocated on the heap. A mapping between class UUIDs and addresses of factory functions is maintained by a data store, which is consulted when objects are instantiated. (ECM also supports "private" classes, whose factories are not registered with the data store, and are instantiated by directly calling the factory function of the class.) The ECM runtime system is accessed through a standard ECM interface, `IShell`, an implementation of which is available at all times. `IShell` is most commonly used to instantiate objects, through `IShell::CreateInstance()`.

ECM comes with an interface description language, known as EIDL, whose syntax is similar to the IDL dialect used with COM, but with ECM-specific keywords. EIDL defines a number of standard types that can be used with ECM classes. ST-Ericsson's software development kit ships with an IDL compiler that for a particular interface can be used to generate C bindings and a "skeleton" C implementation. The latter is a generated class with empty operation bodies, which developers can use as a convenient starting-point when authoring new classes. ECM's C binding is similar to the code presented in Chapter 4.

Classes and arbitrary resource files may be part of *application suites*, which can be built and deployed separately from other suites. They may be installed and uninstalled at runtime, and are made part of a running system through dynamic linking. Application suites are packaged as Native Archive (NAR) files.

From a technical perspective, application suites are in some respects the software components of the ST-Ericsson platform. However, they were not built to support a thriving component ecosystem of third-party components that are composed to build component-based software. Rather, application suites serve as the shared library mechanism of the platform,

[1] ST-Ericsson refers to classes as "components," and to objects as "component instances," but acknowledges in the platform documentation that "components facilitate an object-based development paradigm." Also, the OPA documentation occasionally refers to "component instances" as "objects." For consistency, this thesis refers to these ECM entities as classes and objects, reserving the "component" word for entities that are in most respects compatible with the definitions of Chapter 1.

[2] ECM also includes `IStaticRoot`, which is the root interface of an alternative interface hierarchy. `IStaticRoot` interfaces essentially group together procedural operations, as these interfaces can only have one implementation.

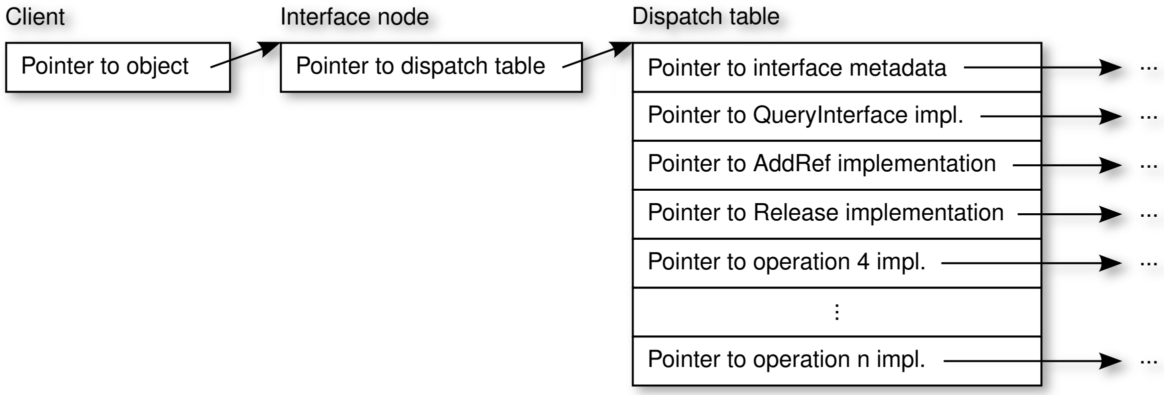
Client
Pointer to object
Interface node
Pointer to dispatch table
Dispatch table
Pointer to interface metadata
Pointer to QueryInterface impl.
Pointer to AddRef implementation
Pointer to Release implementation
Pointer to operation 4 impl.
⋮
Pointer to operation n impl.
···

Figure 6.1 Memory layout of ECM/ECMX interfaces

enabling short build times and a way of preventing unwanted dependencies between suites.

ST-Ericsson offers its customers a choice between linking their application suites dynamically with the platform, as described above, and of linking statically to form a monolith containing both the customer code and the platform. If the latter option is used, ECM is effectively reduced from a combined component and object model to only an object model.

6.2 Enter Sony Ericsson

In the first few years of the new millennium, cellular phone manufacturers like Sony Ericsson added features to their products at a furious pace. The bulky communications devices of the middle of the 1990s developed into slender all-purpose devices, with high-resolution color displays, three-dimensional downloadable games, built-in cameras and music players.

It is no surprise, then, that Sony Ericsson's codebase ballooned in size and complexity. At the time when Ericsson and Sony joined forces, most of the codebase was written in C, and consisted of a mixture of user-facing applications, services running in their own processes and libraries running in the caller's context. Code running in different processes communicated using a technology requiring developers to handle marshalling manually.[3]

In ECM, Sony Ericsson saw the potential to modularize its codebase by making use of ECM's object model for internal projects. However, ECM was found wanting in some areas, prompting Sony Ericsson to devise an extended version, named ECM Extended, or ECMX for short.[4] A number of features have been added to ECMX over the years, including the ability to generate proxies for inter-process communication, a Java language binding, single implementation inheritance, execution tracing and a declarative means of ensuring thread-

[3] Sony Ericsson sells phones that run a variety of operating systems, including Symbian, Windows Mobile and the Linux-based Android. Network access is typically provided by a second chip from ST-Ericsson running Enea's OSE real-time operating system. The work related to Sony Ericsson presented in this thesis invariably concerns the company's so-called "central" phones, which make up the bulk of sales and are primarily developed in Lund, Sweden. The application software of these phones either runs directly on ST-Ericsson's network access CPU, or on a dedicated CPU also running OSE.

[4] The ECMX name is rarely used within Sony Ericsson, and few developers recognize it. Instead, the technology is simply referred to as "IDL." This thesis uses the ECMX name to avoid confusion with other interface description languages and to make the ECM heritage clear.

safety.[5] The first two features are discussed at some length in this section, and execution tracing is the subject of Chapter 7.

ECM and ECMX remain binary compatible, enabling an ECMX class to implement an OPA outgoing interface. However, Sony Ericsson's IDL dialect is no longer fully compatible with EIDL, as ST-Ericsson continued to add new features to its version after ECMX was born. There are plans to alleviate this situation through closer cooperation between Sony Ericsson and ST-Ericsson, with the desired outcome of only having one IDL dialect, one IDL compiler, and one ECM/ECMX runtime system.

Sony Ericsson has traditionally elected to link its software statically with ST-Ericsson's platform. There have been efforts in recent years to separate the software into dynamically linkable modules, primarily to reduce build times and to ensure that there are no unwanted dependencies between modules. This effort is developed in-house, and does not make use of ST-Ericsson's application suites. Dynamically linked modules are required to expose their functionality exclusively through IDL interfaces, thus using the indirection of dispatch tables not only to facilitate dynamic dispatch, but also dynamic linking (as discussed on page 74).

Like ST-Ericsson's application suites, Sony Ericsson's modules in some respects qualify as software components, but only from a technical point of view, as they are not intended to foster a vibrant component ecosystem comprised of third-party components (though they would likely make a good base on which to build a component model).

Sony Ericsson does make use of third-party libraries in its phones (including the Access NetFront web browser and Nuance's T9 predictive text input software), but these libraries are typically shipped by their vendors as pre-compiled libraries bundled with C header files or as (often obfuscated) C source code. Integrating third-party code in this fashion is expensive and labor-intensive, as "glue code" must often be written that gives the third-party code access to the native environment (such as drawing to the screen), or that replaces libraries bundled with the third-party software with calls to pre-existing code with equivalent functionality (a practice which helps cut down on code size).

Sony Ericsson's phones offer one environment, though, that allows for separately deployed third-party extensions that are loaded at runtime and have access to a rich object-oriented library: Java applications. These applications have many of the properties of software components, including being written to vendor-agnostic standards.[6] As applications, though, they must be considered too coarse-grained to be considered true components, much like applications for desktop computers.

6.2.1 Inter-process communication

As early as 1976, White suggested the use of remote procedures to facilitate out-of-process calls, instead of requiring developers to directly send a data stream compatible with a certain network protocol. White's proposal called for a standardized type system used for marshalling, a choice between synchronous and asynchronous calls, and for modifying compilers "to provide minor variants of their normal procedure-calling constructs for addressing remote procedures" (thus generating code for calling remote procedures). The Open Group's Distributed Computing

[5] "Thread-safety" is somewhat of a misnomer when applied to Sony Ericsson's system, as the main use of this declarative attribute is to protect code from concurrent access by multiple processes. The memory management unit (MMU) in Sony Ericsson's OSE-based products is typically not used to protect the memory owned by one process from tampering by other processes, though there are exceptions.

[6] Java applications can often elect to use proprietary extensions, though.

Environment (DCE) introduced an interface description language from which procedural proxies were generated, for use with inter-process and inter-machine communication. Before DCE, developers often wrote tools that generated such proxies directly from C header files (Hludzinski 1998).[7] When Sony Ericsson created a new IDL compiler that could generate proxies for inter-process communication automatically, there was thus ample precedent for this effort.

The proxies in ECMX work similarly to proxies in other systems. Once set up, a client-side proxy masquerades as the remote object, and passes all invocations to the server-side proxy, which invokes the call on the true remote object. The server-side proxy ensures that the return value is transmitted back to the client-side proxy, which returns this value to the client. ECMX does not support a concept similar to COM's in-process handlers, which can elect to handle some operations locally in order to boost performance. (A client-side proxy is simply called a "proxy" in ECMX, and a server-side proxy is called a "stub," which is consistent with the naming convention used by COM.)

Clients normally need not concern themselves with the mechanics of setting up proxies. A server always provides an object, running in the client's context, whose sole purpose is to set up the connection and provide a client-side proxy. Such objects are called *managers*. A manager encapsulates all knowledge required to set up a connection with the server, including the process identifier of the server process.

As part of the build process, proxy classes are generated by the IDL compiler for all interfaces that should be available across process boundaries. (There is not enough type information available at runtime to dynamically synthesize client-side proxies and use generic server-side proxies, a technique exemplified by Java's RMI and .NET Remoting, which are described in Chapter 5.[8]) There is only one client-side proxy class and one server-side proxy class per interface, but every new connection between a client and a server gets freshly created client-side and server-side proxy instances.

In OPA, every process is associated with an object implementing the `IApplication` interface. An OPA process can either elect to process messages sent from other processes itself, or it can let the system handle this aspect by deferring message handling to a system-provided message loop. This message loop routes messages not addressed to a specific recipient to the `IApplication::OnReceivedMsg()` operation of the object associated with the process. Messages processed by the system message loop can also be addressed to a specific *handler*, which is an object implementing the `IHandler` interface, or one of its descendants. Such messages are associated with OPA *sessions*. The system message loop routes messages by consulting a per-process table mapping session identifiers to handler objects. Server-side proxies are OPA handlers, and the system message loop thus routes messages sent from client-side proxies directly to the appropriate server-side proxies.

The code for setting up and tearing down proxy connections is part of the ECMX runtime system. When a connection is set up from a client context, a message is sent to the server

[7] This heritage may go some way toward explaining why many interface description languages, including those used by DCE, COM and ECM/ECMX, are reminiscent of C.

[8] Space is at a premium in most embedded systems, and metadata in the form of runtime-accessible type information does occupy space. However, so does generated code, especially highly repetitive, verbose code typical of proxies for inter-process communication. Including more runtime type information makes it possible to use domain-agnostic system classes and runtime-synthesized objects in lieu of automatically generated classes. This approach may actually use less space overall, and has other benefits as well, such as enabling very late binding and frameworks that rely on naming conventions.

asking it to create a session linked to a newly created server-side proxy. The server sends a message back informing the client of the session identifier, which then creates a client-side proxy, initialized with the session identifier and the process identifier of the server. When a client-side proxy determines that there are no live references to it, it destroys itself, and asks the server-side proxy to do likewise. (Proxies are reference-counted like all other ECMX objects.)

ECMX does not use a formalized wire format. Primitive arguments to an operation, such as integers, are serialized by the client-side proxy simply by putting them in a C structure and copying its contents byte-for-byte into a message. The server-side proxy uses the same C structure to deserialize the primitive arguments.[9] Pointers to arrays and strings are marshaled by copying their contents into the message. (Their runtime size must be given as a separate argument, and this argument must be designated as such in the IDL file.[10])

Interface references can also be marshalled, and are always passed by reference. A server receiving an interface reference accesses the client-side object through proxies in the reverse direction, which are automatically set up by the generated code (illustrated in Figure 2.2 on page 18). If a process `A` sends a local interface reference to a process `B`, which sends this interface reference to both processes `C` and `A`, process `C` will receive a client-side proxy communicating directly with a server-side proxy in process `A`, and process `A` will receive a direct reference to its local interface reference.

ECMX supports both synchronous and asynchronous invocation semantics. A synchronous call is realized by blocking the client process until the server sends a response. In common with other similar systems, such calls are several orders of magnitude slower than direct function calls (an average response time of 350 microseconds, which includes marshalling and context switches, has been observed using an in-circuit debugger). Asynchronous operations allow the client to continue executing while the server processes the request. Passing interface references to a remote object proves especially useful when using asynchronous operations, as it gives the server an opportunity to respond. (There is a pattern for subscriptions, which are used by clients that wish to subscribe to events from a server, such as file system changes.)

Synchronous operations, while convenient, pose a problem in the form of deadlocks (as in most other systems). A process invoking a synchronous operation on a client-side proxy is blocked, and will only resume execution after a response message has been received. If the target process is blocked waiting for a reply from the first process, neither process will resume execution. (An arbitrary number of processes can be involved in a deadlock, if the target process is only indirectly waiting for a response from the first process.)

Deadlocks could be avoided by electing to consider arbitrary messages while awaiting a reply, and not just the expected return message. This is not done for a few reasons, chief among them the desire to avoid reentrancy issues. A client invoking a synchronous operation on an object is not necessarily in a consistent state at the time of the invocation, and considering arbitrary messages could thus lead to unexpected behavior.

The sizes of the call stacks in Sony Ericsson's system are fixed, making it possible to write beyond their confines. Doing so typically results in unintended and likely erroneous behavior, which makes it imperative that stack growth is kept under control. Acting on arbitrary messages while waiting for a response message to a synchronous request would risk writing past the end of the call stack of the currently executing process.

[9] This is only true for classes written in native code. Remote objects written in Java use server-side proxies mostly written in Java. See section 6.2.2.

[10] Null-terminated strings need no argument specifying their size, as the runtime system is capable of gathering this information from the string itself.

6.2.2 Java binding

Sony Ericsson's Java environment allows independently developed applications and games to run on its cellular phones. This environment supports the Connected Limited Device Configuration (CLDC) of the Micro Edition of Java, as well as its Mobile Information Device Profile (MIDP). Sony Ericsson also supports a large number of supplemental specifications, in the form of Java Standardization Requests (JSRs).

In addition to allowing users to add software to their own devices, the Java environment allows Sony Ericsson itself to develop applications and related functionality in Java. Having a standardized application platform also makes it possible to outsource development to other development organizations. At the 2009 JavaOne conference, Sony Ericsson stated that all internal application development will be done in Java in the future (Sun Microsystems 2009).

In order to support internal Java development, there must be a way to access native functionality that goes beyond that offered by the vendor-neutral Java interfaces. A Java binding for ECMX exists for this purpose, which is not only used by Sony Ericsson's applications, but also by some of the company's JSR implementations.

All Java applications in Sony Ericsson's system run in a specialized process, which can run multiple applications simultaneously (it includes an internal scheduler that ensures that Java applications get an equal share of the host's resources). The Java binding enables Java applications to access native services that offer IDL interfaces. Java applications can pass outgoing interface references to operations of native objects, and can thus be notified when events occur, or request information from native objects asynchronously (as well as synchronously). Aside from calling Java objects through outgoing interfaces, native code cannot call on the services of Java objects—all interactions between native and Java code must thus be initiated from Java. Also, while native code can execute in the Java process, Java code always executes in the Java process.

The IDL compiler strives to create Java code that fits in with its environment. `RVoid` return values that signal errors give rise to exceptions, output arguments are mapped to true return values[11] and ECMX types are mapped to the most appropriate Java type—ECMX interfaces are mapped to Java interfaces and pointers to characters are mapped to Java `String` objects, for instance. Java developers need not concern themselves with the `IShell` interface—ECMX objects are instantiated using static methods of generated Java classes that correspond to ECMX classes (there is one static factory method per interface implemented by an ECMX class). Nor do Java developers need to handle reference counting—the system automatically adds a reference to the native object when the Java object representing it is created, and removes it when the Java object is collected by the garbage collector.

Native ECMX objects are represented by Java proxy objects, which implement the generated Java interfaces that correspond to IDL interfaces. A Java proxy object maintains its link to the native object by including its address as part of its instance data. A simple way to enable these proxy objects to call on the services of the native ECMX objects would be to mark the Java methods corresponding to ECMX operations as `native`, and generate C code performing the call at build-time.

Presumably for reasons of space-efficiency, Sony Ericsson instead opts to handle all such calls with a single native function (that is part of the ECMX runtime system). The Java proxy

[11] Multiple output arguments are permissible in the IDL dialect used by ECMX but pose problems for the Java binding (though they are supported). As a result, Sony Ericsson's IDL guidelines warn against this construct.

objects package all arguments in a format easily digestible by this function, which converts them to the format expected by native objects (using the foreign function interface of the Java virtual machine). As the native function does not have compile-time knowledge of the call to invoke, it must invoke the call using code that is call-agnostic, which is not possible in C. Hence, this part is written in architecture-specific assembly code, which manually pushes a stack frame containing the arguments onto the call stack, and jumps to the implementation of the native operation. If the native object executes in the client's context, control is passed directly to the native object, executing in the Java process. If the native object runs in a different process, the object that control is passed to is a client-side proxy object that forwards the call to a server-side proxy object.

Enabling Java code to be called from native code is straight-forward, as Java code always runs in the Java process. Had it been possible to run Java code in the caller's native process, outside of the Java process, an ECMX object would have had to be synthesized at runtime, with an implementation forwarding all calls to Java using the foreign function interface of the Java virtual machine (as suggested in section 5.4.1). As Java code is restricted to the Java process, all Java calls from native processes are realized using inter-process communication. The native process invokes operations on a native ECMX object, a client-side proxy, which sends messages to the Java process.

The Java process runs its own message loop and can, with the help of the ECMX runtime system, route messages directly to Java objects—*handlers*—that have previously expressed an interest in handling certain messages. The server-side proxy that receives messages sent by a native client-side proxy is thus written in Java, and marshals arguments with the help of native functions in the ECMX runtime system, after which it invokes the call on the target Java object. The ability of the Java process message loop to forward messages directly to handlers is analogous to what the OPA message loop does for native processes.

CHAPTER 7

Implementing interception

Declarative programming is commonly understood to be concerned with telling the computer *what* to do, and not *how* it should go about doing it. Imperative programming, on the other hand, entails writing statements that inform the computer of the exact steps needed to accomplish a certain task. It is becoming increasingly common to mix imperative programming with declarative attributes. Such programs are at their heart still imperative, but certain aspects of them are described declaratively.

Component technology, especially as used in enterprise settings, has been at the forefront of this movement. Many component models enable programmers or administrators to configure components declaratively, using either custom metadata written in the programming language that the component is implemented in, specialized tools for administrators or separate configuration files (often written in an XML-based language). Describing an aspect declaratively enables a runtime environment to provide services that would otherwise have to be called upon manually. This is especially useful for enterprise software, that would otherwise have to include error-prone and repetitive code. Modern component models enable many of the services enumerated in section 1.7 through declarative attributes.

One aspect that is commonly expressed using declarative attributes is threading. An object that keeps no state outside of the arguments its operations receive, and is not otherwise dependent on globally accessible resources, is both reentrant and thread-safe, and can be used concurrently by several threads with no ill effects. An object that is not reentrant may ensure that it is safe to access from multiple concurrently executing threads by keeping its state in a data store specific to the currently executing thread (so-called *thread-local storage*). Some objects are inherently not thread-safe, and it is vital that only one thread at a time is allowed to enter their operation bodies. The classic way to achieve the latter is to manually ensure that threads wait their turn, using a construct such as a counting semaphore. A declarative attribute related to threading enables a developer to state whether an object is thread-safe, and rely on the component model implementation to enforce the desired threading policy.

COM, for instance, subdivides a process into *apartments*, which are concurrency boundaries that implement different threading policies. Objects belong to apartments that cater to their threading characteristics. COM ensures that only one call at a time reaches a non-thread-safe object by insisting that inter-apartment calls are made using proxies (thus serializing invocations). As a result, calls to non-thread-safe objects are converted to messages, which are put in a message queue to be inspected at the leisure of the thread servicing the non-thread-safe objects. Only one thread is allowed in an apartment that houses non-thread-safe objects (Prosise 2001).

Database transactions are also a popular target for declarative attributes. Instead of manually having to commit or roll back a transaction, an object that is part of a context that participates in a transaction can have the component model implementation perform this service for it—committing the transaction if no exception is thrown, and rolling it back otherwise. Component model implementations are typically able to manage transactions that span multiple database servers. Microsoft introduced their Microsoft Transaction Server (MTS) product, built on top of COM, to provide services such as transaction processing to components running in its application server. MTS was later merged with COM to form COM+. Declarative transaction processing is provided by many other environments for the enterprise, such as CORBA, .NET, and the Enterprise Edition of the Java platform.

Component models provide these services by intercepting calls to objects. When a call has been intercepted, the component model implementation executes code specific to the requested service before passing control to the target object, if such access has not been barred by a security service. Code may also be executed after the commencement of a call (for instance, to commit a transaction if no exception has been thrown). This mechanism is similar to aspect-oriented programming in that the services provided by component models can be likened to aspects.

7.1 Interception practices

Intercepting calls to statically bound native functions generally involves patching code at runtime, in much the same way software debuggers catch control breakpoints.[1] Hunt and Scott (1999) suggest moving the first few instructions at the start of an intercepted function to a "trampoline," and replacing them with a jump instruction which transfers execution to the trampoline. This code can take any actions it desires before executing the copied instructions from the original function and transferring execution back to the remainder of it. Hunt and Scott suggest intercepting the return call by overwriting the return address on the call stack, and storing this value in thread-local storage for later reference.

Due to the characteristics of the component technology studied in this work, interception can be implemented in ways that do not involve patching existing code at runtime. This is especially beneficial for embedded systems that may run code directly from read-only or flash memory, and may thus not allow for the runtime modification of code. Component models that mandate the use of runtime software to invoke operations can implement interception trivially. A CORBA ORB, for instance, allows interceptors to register with it, and ensures that they are invoked when calls are made (Schmidt and Vinoski 2003). Platforms based on virtual machines, such as Java and .NET, can easily implement interception, as the "machines" they target are software constructs (unless ahead-of-time compilation to native code is employed). The Enterprise Edition of Java, though, elects to implement interception by requiring that Enterprise JavaBeans are accessed only through an explicit intermediary that invokes services; clients are thus under no pretense that they are communicating directly with the target object (Szyperski et al. 2002:310).

[1] Calls to shared library functions are easier to intercept as they typically go through a jump table which can be patched instead of the code itself. The one caveat is that this only works for implicit runtime linking.

Component models based on binary standards, such as COM, allow objects to communicate directly without the use of a mediating runtime system, and must therefore use different means to realize interception. Due to the universal use of dynamic dispatch by the component models considered in this thesis, this can be done without patching code at runtime, as dynamic dispatch implies that binding to an implementation is done at runtime. The implementation may thus not be the true target object, but a wrapper that forwards calls and ensures that the services provided by the component model are invoked before and after the call to the target object is made. (Such a wrapper object is also free *not* to forward requests, at its own discretion.)

The simplest way to achieve this scheme is arguably to statically generate wrapper classes, in much the same way that proxy classes are generated at build-time in Sony Ericsson's ECMX system. The mechanism used to instantiate objects needs to be wrapper-savvy and return a wrapper object if one is needed. However, generating wrapper classes at build-time adds unnecessary code bloat, which can be avoided by synthesizing wrapper objects at runtime.

Brown (1999a,b) does just this by introducing a generic wrapper that can wrap any object, and allows for the execution of arbitrary code before and after the operations of the target object are invoked. This generic wrapper is interesting in that it does not require type information to be available at runtime (which also means that services cannot usefully process arguments given to operations). It manages this feat by using a domain-agnostic dispatch table, whose functions delegate calls to a single generic function that dispatches the call to the original object and allows services to run. The only information needed by this generic function is the offset into the dispatch table, which the functions pointed to by the generic dispatch table helpfully push onto the call stack before invoking the generic function. (By necessity, all this code must be written in assembly language.) The generic wrapper intercepts return calls from the target object by overwriting the return address in the stack frame.

The one weakness of the generic wrapper is that interceptors cannot prevent an invocation from propagating to the target object, which is due to the calling convention used by COM on 32-bit systems—`stdcall`. This calling convention requires the receiver of calls to adjust the stack pointer. As a result, the interceptor must let all invocations propagate so that the receiver can adjust the stack pointer, as it cannot do this on its own due to lack of type information. Incidentally, this is not an issue on 64-bit Windows systems, as the one universal calling convention dictates that the caller adjusts the stack pointer.

Brown's generic wrapper must be explicitly instantiated at runtime by users. By contrast, wrappers are automatically used by the COM-based interception system devised by Hunt and Scott (1999). This is realized by intercepting all object-instantiation calls to COM's shared library, and returning wrappers instead of the sought objects. These functions are intercepted by patching parts of the COM runtime system, as described on the preceding page. Arbitrary services may be implemented on top of this system.

Microsoft Transaction Server uses a similar system, but allows only for system-provided services. It ensures that its wrappers are used, not by patching COM's object-instantiation functions, but by modifying the Registry, which normally associates classes with the files that house them. These modifications ensure that a component provided by MTS is identified as the file housing classes of interest to MTS. It maintains an alternative data store containing the real file names, thereby ensuring that its wrappers can instantiate the true target objects when they are created. This data store also contains information on what services should be provided to objects. Developers must take care not to return a direct reference to an MTS object, as doing so circumvents the interception system; all external invocations must go through wrappers (Pattison 2000).

With the release of Windows 2000, Microsoft brought a number of improvements to COM in the form of COM+. The most significant change was merging the functionality of MTS with COM. As a result, COM gained an application server as well as enterprise services configured using declarative attributes and realized using interception. Whereas MTS had been layered strictly on top of COM, COM+ integrated the enterprise features directly.

In COM+, all objects reside in *contexts*, which themselves are part of apartments. Objects that are similarly configured may be part of the same context, in which case they are able to access one another directly. Objects that are part of different contexts access one another through proxies, which act as the wrappers that allow COM+ to intercept invocations. All object references are specific to the context in which it was created—sharing an object reference with an object in a different context prevents COM+ from intercepting calls (Box 1999).

Like MTS, COM+ maintains a data store separate from the Registry to store the declarative attributes of classes. Unlike MTS, it does not need to modify the Registry to make it point to COM+ wrappers, as the COM+ runtime system is service-savvy. With COM+, interception has been integrated directly with COM, simplifying the technology significantly.

7.2 Implementing execution tracing at Sony Ericsson

Execution tracing allows developers and administrators to monitor the runtime behavior of programs. Some tracing facilities monitor some or all of the actions taken by the program for a limited time, while others allow developers to directly or indirectly insert *tracepoints* into the program code. When encountered at runtime, they collect information for later perusal or let the user know that the tracepoint has been encountered (say, by writing to a file or to standard output). Tracing is notably different from debugging, where breakpoints stop the execution of the program.

There are many examples of trace facilities in industry. In their simplest form, developers manually add “print statements” to their programs, often at the start and end of functions or methods.[2] The `ltrace` and `strace` programs, that are often part of Linux distributions, print information to standard output when the instrumented program calls functions in shared libraries and invokes system calls, respectively. For embedded systems, in-circuit debuggers can often generate precise traces of all activity of an embedded processor, which coupled with metadata emitted by the linker can yield insights into the system behavior. Emulators, such as QEMU, often have facilities to generate traces, which they can easily do as they have complete control of the execution environment.

A prominent tracing facility in industry is DTrace, by Sun Microsystems. It allows developers to run scripts that collect data when tracepoints are encountered, and supports both kernel-mode and user-mode tracing. DTrace tracepoints can be enabled and disabled at runtime. Most disabled tracepoints have no performance cost, which DTrace realizes by rewriting code at runtime in much the same way as described on page 104. Scripts are designed to be safe to run even in a production system, and are therefore written for a “safe” virtual machine. An example of this safety property is that the instruction set of the virtual machine only allows forward jumps, making constructs such as non-terminating loops (and indeed any loops) impossible to express (Cantrill et al. 2004).

[2] Some compilers can automate this, by calling pre-defined functions whenever a function or method body is entered or exited. GNU’s GCC, for instance, provides the `-finstrument-functions` option for this purpose.

Listing 7.1 Excerpt from a sample trace file

```
C98487__2c434f7434124CJPEGImageFactory__
E98487__2c434f7434200CJPEGImageFactory_IUIImageFileFactory_CreateImage
C98487__2c56cc7c34264CJPEGImage__
L98487__2c434f7434404CJPEGImageFactory_IUIImageFileFactory_CreateImage
D98487__2c434f7434436CJPEGImageFactory__
E98487__2c56cc7c34484CJPEGImage_IUIImage_GetDimensions
L98487__2c56cc7c34528CJPEGImage_IUIImage_GetDimensions
E98487__2c56cc7c34576CJPEGImage_IUIImageJPEGSettings_HasThumbnail
L98487__2c56cc7c34668CJPEGImage_IUIImageJPEGSettings_HasThumbnail
```

A simple execution tracing facility for ECMX objects, realized using code interception, has been implemented as part of this thesis work. Developers select the classes whose objects should be traced before building the product, and can easily select all classes that belong to a particular module, all classes whose names match a certain pattern, or even all classes present in the system.

The instantiation and destruction of objects, as well as all occurrences of code entering and leaving their operation bodies, are written to a human-readable trace file that if possible resides on an external memory card (meaning that its size is only limited to the capacity of the memory card). Entries in the trace file include a timestamp and the name of the class, and for entries that record code entering or leaving operations, the name of the interface and operation. Crucially for debugging calls involving multiple processes, entries also include the process identifier of the currently running process. Operations belonging to proxies are specially identified, and such entries also include the process identifier of the process on the receiving end.

Listing 7.1 contains a heavily abridged version of a trace file captured at Sony Ericsson. Entries in the trace file, which correspond to lines, are recognized by a finite automaton that corresponds to the following regular expression (Perl syntax):[3]

```
[CDELPAS]\d+_\d*_\w{8}\d+[\w\d]+_[\w\d]*_[\w\d]*
```

The first character identifies the type of the entry. **C** indicates that a class has been instantiated and **D** that an object has been destroyed. **E** signifies that an operation body has been entered and **L** that one has been left. **P** denotes that the body of a synchronous operation that is part of a client-side proxy has been entered and **A** likewise for the body of an asynchronous operation. Finally, **S** signifies that the body of an operation belonging to a server-side proxy has been entered (a “stub,” in Sony Ericsson parlance).

The number that follows is the process identifier of the currently running process. Entries that reflect proxy operations (identified by **P**, **A** or **S**) next include the process identifier of the receiving process (this part is blank for the other entry types). The hexadecimal number that

[3] While a finite automaton corresponding to this regular expression successfully matches entries in trace files, the expression is too lenient in that it recognizes some lines that are not quite proper. In particular, object identifiers must be hexadecimal numbers and not arbitrary strings, and class and interface identifiers, as well as names of operations, must not start with a number. A stricter, but less readable, regular expression follows: `[CDELPAS]\d+_\d*_[0-9A-Fa-f]{8}\d+(?:\w[\w\d]*)_(?:\w[\w\d]*)?_(?:\w[\w\d]*)?`

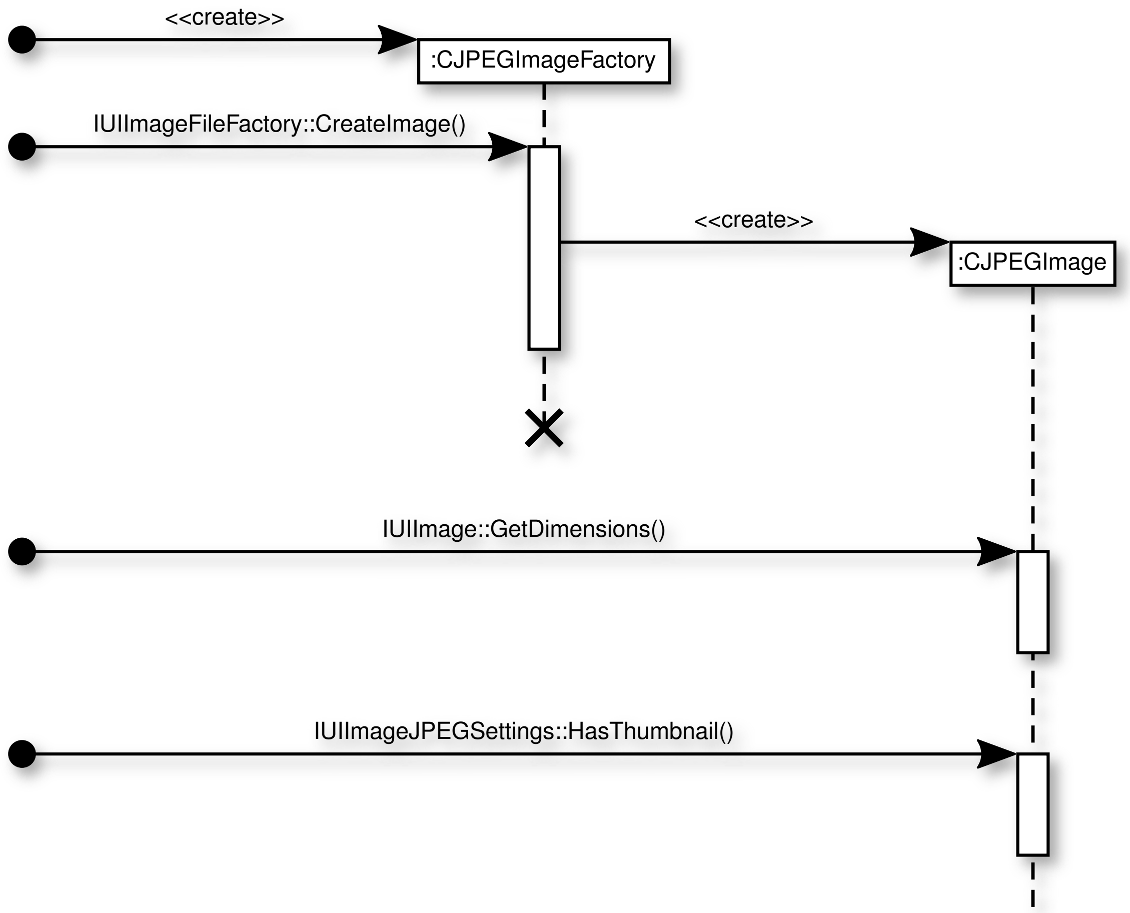
<<create>>
:CJPEGImageFactory
IUIImageFileFactory::CreateImage()
<<create>>
:CJPEGImage
IUIImage::GetDimensions()
IUIImageJPEGSettings::HasThumbnail()

Figure 7.1 UML sequence diagram generated from a sample trace file excerpt

follows uniquely identifies the object that the entry refers to, and is followed by a timestamp.[4] The next string is the compile-time name of the class. The two remaining strings are blank for entries that reflect object instantiation or destruction, but are otherwise the compile-time name of the interface that the called operation belongs to, followed by the name of the called operation.

Also as part of this thesis work, a tool has been written that infers a UML interaction sequence diagram from a trace file. Figure 7.1 shows the (lightly edited) sequence diagram generated from the trace file of Listing 7.1.[5]

This tracing facility is useful for debugging all object interactions, but especially those that cross process boundaries. Entries for proxy operations include the process identifiers of

[4] The object identifier can be any number that uniquely identifies an object, and the only constraint put upon the timestamp is that the higher the number, the later the event. In the current implementation, the object identifier is simply the address of the instance data of the object, and the timestamp is a number specific to the underlying operating system that reflects the time passed since the system was started.

[5] The sequence diagram of Figure 7.1 has been slightly altered to make it consistent with the visual style of the other figures in this thesis. In particular, the font has been changed, a drop shadow has been added, and minor visual glitches have been attended to.

both the sending and receiving processes, making it possible to identify process deadlocks.

COM+ provides a service that corresponds to this tracing facility in the form of COM+ Instrumentation. This service publishes events to clients using a loosely-coupled event service. Clients can subscribe to a wide variety of different events, including notifications related to security, transactions, exceptions and object pools, and notably also object instantiation and destruction, as well as when operations are entered and exited.

The trace feature developed as part of this thesis work has been integrated into Sony Ericsson's main codebase, and is available to all developers. The proxy-related entry types were implemented by Sony Ericsson after the commencement of this thesis work.

7.2.1 Generating UML sequence diagrams

A number of output formats were considered for the tool generating UML sequence diagrams, including those used by leading CASE tools.[6] In the end, the XML-based Scalable Vector Graphics (SVG) format was selected because of its ubiquity and openness. A cursory search at the end of 2005 revealed that no suitable pre-existing sequence diagram generators were available, thus necessitating the creation of a custom solution.

The initial plan called for writing a Perl script converting trace files to a standardized XML-based format for expressing program behavior at runtime. This data would then be converted to SVG using an XSLT stylesheet. This approach was abandoned, as no such standardized format was found other than OMG's XML Metadata Interchange (XMI), and this format was deemed too complex. The tool was instead fully written in Java.

The sequence diagram generator is structured much like a compiler. The front-end parses trace files, and returns an intermediate representation from which an output document is generated by a back-end. To parse the input data, the front-end uses Java's built-in support for regular expressions, with an expression similar to the one presented in the footnote on page 107.

To infer caller–callee relationships, the front-end maintains a call stack—when operation bodies are entered, an entry is pushed onto the call stack, and popped off when the body is exited. When an entry that signifies that an operation is entered is encountered in the trace file, the operation at the top of the call stack is assumed to have called it. Likewise, when an object creation entry is encountered, the operation at the top of the call stack is assumed to have created it. As not all objects are necessarily part of a trace file, these links are somewhat tenuous in that they are at times indirect. For instance, an operation identified as instantiating a certain object may not have created the object directly, it may rather have called on the services of an object not appearing in the trace file.

There are two back-ends available, one that prints a text-only version of the trace file to standard output and one that generates SVG files. The text-only back-end is mostly used for debugging the generator itself. Its output is significantly more readable than a raw trace file, though, due to its use of indentation to convey call depth.

The SVG back-end uses the open-source Apache Batik library, which allows programs to draw on a standard graphics canvas instead of generating SVG directly—Batik converts the graphics primitives to SVG. The SVG back-end does not attempt to have the Y axis of diagrams reflect real time, as doing so would result in very large diagrams.

[6] *Computer-aided software engineering* (CASE) is an umbrella term for software tools that help organizations develop software. It is most commonly used to refer to tools that help with modeling and design. CASE tools typically help developers model a domain visually using UML notation.

7.2.2 Tracing invocations

The tracing facility developed as part of this thesis work can only be enabled at build-time—there are no provisions for enabling and disabling tracepoints at runtime. Unlike the interception techniques described on page 106, this solution does not use wrapper objects that masquerade as the target objects and forward invocations. Instead, the IDL compiler emits code that realizes tracing for traced classes.

In theory, ECMX is a binary standard, and developers can thus write classes without relying on Sony Ericsson's IDL compiler. In practice, Sony Ericsson's IDL dialect is used pervasively, and developers never write ECMX classes without using the IDL compiler—indeed, the technology is simply known as "IDL" in developer vernacular. This is in stark contrast to COM, where an arbitrary number of IDL compilers and other such tools, created by companies other than Microsoft, create COM-compatible classes. The binary standard is the common denominator that binds these disparate solutions together. At Sony Ericsson, though, the sole IDL compiler, which is under the direct control of the company, is always used to create ECMX-compatible classes. IDL is thus a *de facto* standard for creating ECMX classes at Sony Ericsson, if not a *de jure* standard. As such, modifying the IDL compiler to emit tracing code is a workable solution.

Sony Ericsson's IDL compiler creates a number of files when creating a C binding from an IDL file. Of these, only the file housing the domain-specific implementation of a class is expected to be stored in a version-controlled repository. The other files, containing "plumbing" code such as proxies for inter-process communication, UUIDs and notably dispatch tables, are generated by the build system on-demand, and are stored in a directory dedicated to automatically generated files.

The latter property is exploited by this tracing facility to make already-written classes write their runtime behavior to disk, with no changes required to the developer-maintained class source code. When the IDL compiler encounters a class for which tracing has been enabled, it does not point the generated dispatch tables of the class directly to the implementation functions as it ordinarily does—instead, it points the dispatch tables to generated wrapper functions. These functions call on the services that have been enabled declaratively before and after the original function is invoked. (ECMX also uses these wrappers to enable declarative thread-safety.) Tracing class instantiation and object destruction is straight-forward, as the class factory and implementation of `IRoot::Release()` are always provided by the system when the IDL compiler is used.

A sample trace wrapper generated by the IDL compiler is shown in Listing 7.2. The `CSystemTrace` class is part of the ECMX runtime system, and is responsible for writing trace entries to disk.[7] Instances of this class are protected from concurrent access using ECMX's support for declarative thread-safety. Return values from the `CSystemTrace` operations are purposefully thrown away, as a failure to write a trace entry to disk should not prevent the original function from being called, and should not cause the caller to receive a failure code.

An ECMX IDL file ordinarily only contains one class. The IDL compiler generates code that traces invocations when invoked with the command-line argument `--trace`. The build

[7] `CSystemTrace` is an example of a *static* class, which is yet another addition to ECMX. Macros of the form `[class-name]_[interface-name]_[operation-name]` are generated by the IDL compiler for these classes, and expand to calls to system-wide object instances (which are created if necessary). The first version of this work put the required code inline in the wrapper functions themselves, instead of calling on the services of a separate object. After the commencement of this thesis work, Sony Ericsson refactored the tracing facility, and in doing so, created the `CSystemTrace` class.

Listing 7.2 Sample trace wrapper

```
static RVoid
  CTestClassBase_ITestInterface_TestOperation(
    ITestInterface* pITestInterface)
{
  CTestClassBase_t* pThis = (CTestClassBase_t*)pITestInterface->pData;
  RVoid _result;

  CSystemTrace_ISystemTrace_EnterOperation("CTestClass",
                                           "ITestInterface",
                                           "TestOperation",
                                           (FUint32)pThis);

  _result = CTestClass_ITestInterface_TestOperation(pITestInterface);

  CSystemTrace_ISystemTrace_LeaveOperation("CTestClass",
                                           "ITestInterface",
                                           "TestOperation",
                                           (FUint32)pThis);

  return _result;
}
```

Listing 7.3 Sample configuration file enabling tracing

```
[SourceFiles_Options]
CFirstTestModuleTestClass.idl +IDL=(--trace)   # Trace a specific class.
CSecondTestModule*.idl +IDL=(--trace)          # Trace a specific module.
#C*.idl +IDL=(--trace)                         # Trace all classes.
```

system used by Sony Ericsson allows for temporary modifications to the build configuration using a file, `DescrExtra`, that is never checked into the versioned-controlled repository. Using this file, developers can easily add the `--trace` argument to one or several IDL files housing classes. (IDL files should always be named after the classes or interfaces that reside within them.) `DescrExtra` also supports wildcards, making it easy to enable tracing for multiple files. As classes are normally prefixed with the name of the module they are part of, the wildcard feature makes it easy to enable tracing for all classes that are part of a specific module. Listing 7.3 shows a sample `DescrExtra` file.

Sony Ericsson's IDL compiler is written in C++, and its front-end uses GNU Flex for lexical analysis and GNU Bison for generating a parser from a context-free grammar. The intermediate representation consists of an in-memory syntax tree, which the back-end traverses using the visitor design pattern (Gamma et al. 1995). This work has entailed making a number of changes primarily to the visitors of the compiler.

It is debatable whether this tracing facility truly uses interception. The source code of classes is essentially modified in such a way that objects are coerced into writing their own behavior to disk. At runtime, there is no need to intercept calls using separate wrapper objects, as the objects themselves realize tracing.

7.2.3 Future work

Trace files tend to become quite voluminous, especially if tracing is enabled for many classes. As a result, the resulting UML sequence diagrams become very large and consequently hard to read. One way to make diagrams more compact is to represent redundant data more efficiently. Taniguchi et al. (2005) propose a number of compaction rules that make UML sequence diagrams generated from traces more concise, including the removal of repetitions. The authors report that sample traces were considerably reduced in size when their compaction rules were applied to them, ranging from one hundredth of the original size to seven percent of it. The work of Taniguchi et al. could be profitably applied to the UML sequence diagram generator developed as part of this thesis work.

UML sequence diagrams generated by this tool must be viewed using an external viewer. It might be worthwhile to create a dedicated viewer application allowing the user to filter out specific classes.

It is not clear that UML sequence diagrams are the best way to visualize program traces. Renieris and Reiss (1999) present some alternatives, including a spiral view and a space-efficient linear view that uses the horizontal axis to represent (real) time, the vertical axis to represent call depth and color to represent the called function. A viewer application for traces could allow the user to select between several different visualizations.

The data contained in trace file may not only be used to visualize program behavior, it can also be used for performance analysis. The timestamps present in trace files are not used by the UML sequence diagram generator, but could fruitfully be used by a tool reporting on performance, in effect creating a simple profiler.

The primary benefit of having objects log their own behavior, by redirecting their dispatch table entries to statically generated wrapper functions, is simplicity. There are two significant disadvantages, though. First, source code must be generated statically, leading to an increase in code size. Also, the product must be rebuilt to enable or disable the tracing facility for certain classes, an annoyance in an environment with long build times. (This issue will be significantly alleviated once Sony Ericsson's efforts to introduce dynamically linked modules get underway, though, as build times will be substantially reduced.) Second, the current tracing facility can not support tracing calls to all objects implementing a certain interface—the trace feature is enabled on a per-class basis only.

These issues could be solved by introducing wrapper objects instead of modifying the source code of classes, the only downside being the added implementation complexity. This would make the implementation of the tracing facility similar to the interception system proposed by Hunt and Scott (1999), and would allow for the easy integration of additional services.

Had wrappers been available, the implementation of `IShell::CreateInstance()` and `IRoot::QueryInterface()` would have to be interception-aware, in that these operations would have to determine whether to return direct references or references through wrapper objects. Making `IRoot::QueryInterface()` interception-aware would be straight-forward, as wrapper objects need only be returned from other wrapper objects, and these objects are free to use any implementation of this operation that they desire. The implementation of `IShell` is provided by ST-Ericsson, and while patching ST-Ericsson's source code is possible, it is not desirable. It would be possible to intercept calls, though, which again is similar to the approach taken by Hunt and Scott.

To make declarative attributes, such as tracing, available at runtime, a separate data store would have to be created (the current data store containing class UUIDs and their mappings is part of ECM, and is shipped as part of ST-Ericsson's platform). To replicate the ease-of-use of the current tracing facility, a means of setting declarative attributes at build-time (using compile-time names) would have to be implemented.

Wrapper classes could be generated at build-time, much like proxy classes, but this would waste space unnecessarily. As demonstrated by Brown (1999a,b), synthesizing wrapper objects at runtime that implement arbitrary interfaces is possible without access to type information.

The current tracing facility manages to include compile-time names in trace files for the simple reason that these names are embedded as string literals in the generated wrapper functions. Statically generated wrapper objects would be generated on a per-interface basis, and while they could include the compile-time names of interfaces and operations, they would not be able to write the compile-time class name to a trace file. If wrapper objects are to be synthesized at runtime, no such information is available, as the synthesized implementation is completely class-agnostic. In order to write trace files in the format exemplified by Listing 7.1, more complete runtime type information would have to be available.

An alternative would be to forego creating human-readable trace files, and instead use runtime names and dispatch table indices. Instead of including the compile-time names of classes and interfaces, as well as human-readable operation names, a trace entry would instead use a class UUID, interface UUID and the index of the operation in the dispatch table. (Indeed, this is the information reported by COM+ Instrumentation.) Such a trace file would be significantly more compact than the current format, and could be post-processed on a computer with access to the source IDL files to create a human-readable trace file.

Another advantage to moving to a runtime solution with dynamically synthesized wrapper objects, instead of statically generated wrapper functions, would be that tracing could be enabled and disabled at runtime. ST-Ericsson provides a tool named Interactive Debug, which allows developers to interact with a running system from a PC.[8] This tool could be used for this purpose. Absent runtime type information, developers would have to use runtime names in the form of UUIDs, though.[9] Enabling or disabling tracing for yet-to-be-instantiated objects would be easy, doing likewise for already-existing instances would be considerably harder, as wrappers would then essentially have to be used for all objects, at all times.

Implementing a generic interception system for ECMX, in the manner described above, would be a major undertaking. It would enable, though, not just better support for execution tracing, but the possibility of easily introducing a wealth of additional declarative services to Sony Ericsson's system.

[8] Interactive Debug appears much like a file system to the developer, and provides a hierarchically organized information space that the developer can interact with. It is somewhat similar to the `procfs` virtual file system, which allows users to interact with the running kernel of many Unix-like systems.

[9] As type information is available on the PC, it would be possible to write a tool allowing developers to use compile-time names, that would simply translate these names to runtime names before interacting with the embedded system.

Bibliography

Markus Aleksy, Axel Korthaus and Martin Schader. *Implementing Distributed Systems with Java and CORBA*. Springer-Verlag, 2005.

Jeff Alger. OpenDoc vs. OLE. *MacTech*, 10(8), 1994. Available at `http://www.mactech.com/articles/mactech/Vol.10/10.08/OpenDoc2/`.

Felix Bachmann, Len Bass, Charles Buhman, Santiago Comella-Dorda, Fred Long, John Robert, Robert Seacord and Kurt Wallnau. *Volume II: Technical Concepts of Component-Based Software Engineering, 2nd Edition*. Technical report, Carnegie Mellon Software Engineering Institute, May 2000. Available at `http://www.sei.cmu.edu/pub/documents/00.reports/pdf/00tr008.pdf`.

David Bank. The Java saga. *Wired Magazine*, 3(12), December 1995. Available at `http://www.wired.com/wired/archive/3.12/java.saga.html`.

Neil Bartlett. OSGi in practice, January 2009. Draft preview, no publisher announced. Available at `http://neilbartlett.name/blog/osgibook/`.

Grady Booch, Hedley Apperly, William T. Councill, Martin Griss, George T. Heineman, Ivar Jacobson, Steve Latchem, Barry McGibbon, Davyd Norris and Jeffrey Poulin. The near-term future of Component-Based Software Engineering. In George T. Heineman and William T. Councill, editors, *Component-Based Software Engineering: Putting the Pieces Together*, pages 753–777. Addison-Wesley, 2001.

Don Box. Windows 2000 brings significant refinements to the COM(+) programming model. *Microsoft Systems Journal*, 14(5), May 1999. Available at `http://www.microsoft.com/msj/0599/complusprog/complusprog.aspx`.

Keith Brown. Building a lightweight COM interception framework, part I: The Universal Delegator. *Microsoft Systems Journal*, 14(1):17–29, January 1999a. Available at `http://www.microsoft.com/msj/0199/intercept/intercept.aspx`.

Keith Brown. Building a lightweight COM interception framework, part II: The guts of the UD. *Microsoft Systems Journal*, 14(2):49–59, February 1999b. Available at `http://www.microsoft.com/msj/0299/intercept2/intercept2.aspx`.

Bruce Bukovics. *.NET 2.0 Interoperability Recipes*. Apress, 2006.

Jon Byous. *Java Technology: The Early Years*. Sun Microsystems, 1998. Available at `http://java.sun.com/features/1998/05/birthday.html`.

Charlie Calvert. *Charlie Calvert's Delphi 4 Unleashed.* Sams Publishing, 1999.

Bryan M. Cantrill, Michael W. Shapiro and Adam H. Leventhal. Dynamic instrumentation of production systems. In *Proceedings of USENIX '04*, 2004. Available at `http://www.sun.com/bigadmin/content/dtrace/dtrace_usenix.pdf`.

Alan F. Chalmers. *What is this Thing Called Science?: An Assessment of the Nature and Status of Science and Its Methods*, chapter Theories as structures I: Kuhn's paradigms. University of Queensland Press, 1999.

Stephen Clamage. *Stability of the C++ ABI: Evolution of a Programming Language.* Sun Microsystems, 2002. Available at `http://developers.sun.com/solaris/articles/CC_abi/CC_abi_content.html`.

John Corwin, David F. Bacon, David Grove and Chet Murthy. MJ: A rational module system for Java and its applications. In *Proceedings of the OOPSLA '03 Conference*, pages 241–254. Association for Computing Machinery SIGPLAN, 2003.

Edsger W. Dijkstra. The humble programmer. *Communications of the ACM*, 15(10):859–866, October 1972. Available at `http://www.cs.utexas.edu/users/EWD/ewd03xx/EWD340.PDF`.

Bruce Eckel. *Thinking in Java*, chapter Appendix A: Using non-Java code. Prentice Hall, first edition, 1998. Chapter written by Andrea Provaglio. Available at `http://www.mindviewinc.com/Books/`.

Guy Eddon and Henry Eddon. Understanding the DCOM wire protocol by analyzing network data packets. *Microsoft Systems Journal*, 13(3), March 1998. Available at `http://www.microsoft.com/msj/0398/dcom.aspx`.

Hans-Erik Eriksson, Magnus Penker, Brian Lyons and David Fado. *UML 2 Toolkit*, chapter Representing Architecture, pages 255–257. John Wiley and Sons, 2004.

Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley, 1995.

Angana Ghosh, Magnus Olsson and Patrik Persson. Ericsson Review: Open application environments in mobile devices: Focus on JME and Ericsson Mobile Platforms. *Ericsson Review*, 82(2), 2005. Available at `http://ericsson.com/ericsson/corpinfo/publications/review/2005_02/files/200502.pdf`.

Crispin Goswell. *The COM Programmer's Cookbook.* Microsoft, 1995. Available at `http://msdn.microsoft.com/en-us/library/ms809982.aspx`.

George T. Heineman and William T. Councill, editors. *Component-Based Software Engineering: Putting the Pieces Together.* Addison-Wesley, 2001a.

George T. Heineman and William T. Councill. Definition of a software component and its elements. In George T. Heineman and William T. Councill, editors, *Component-Based Software Engineering: Putting the Pieces Together*, pages 5–21. Addison-Wesley, 2001b.

George T. Heineman and William T. Councill. Summary. In George T. Heineman and William T. Councill, editors, *Component-Based Software Engineering: Putting the Pieces Together*, pages 741–753. Addison-Wesley, 2001c.

Michi Henning. The rise and fall of CORBA. *ACM Queue*, 4(5):28–34, June 2006. Available at `http://queue.acm.org/detail.cfm?id=1142044`.

Bill Hludzinski. Understanding Interface Definition Language: A developer's survival guide. *Microsoft Systems Journal*, 13(8), August 1998. Available at `http://www.microsoft.com/msj/0898/idl/idl.aspx`.

Galen C. Hunt and Michael L. Scott. Intercepting and instrumenting COM applications. In *Proceedings of the 5th Conference on Object-Oriented Technologies and Systems (COOTS'99)*, pages 45–56, 1999.

Dong-Heon Jung, JongKuk Park, Sung-Hwan Bae, Jaemok Lee and Soo-Mook Moon. Efficient exception handling in Java bytecode-to-C ahead-of-time compiler for embedded systems. *Computer Languages, Systems and Structures*, 34:170–183, 2008.

Michael Kornby. The EMP story. *Ericsson Review*, 82(1), 2005. Available at `http://ericsson.com/ericsson/corpinfo/publications/review/2005_01/files/2005013.pdf`.

Joseph C. Libby and Kenneth B. Kent. An embedded implementation of the Common Language Infrastructure. *Journal of Systems Architecture*, 55:114–126, 2009.

Tim Lindholm and Frank Yellin. *The Java Virtual Machine Specification*, chapter Loading, Linking, and Initializing. Prentice Hall, second edition, 1999. Available at `http://java.sun.com/docs/books/jvms/second_edition/html/VMSpecTOC.doc.html`.

Ray Lischner. *Delphi in a Nutshell: A Desktop Quick Reference.* O'Reilly & Associates, 2000.

Steve Lohr. Microsoft told to stop shipments of software at issue in rival's suit. *The New York Times*, 1998. Published on the 18th of November.

Frank Lüders, Ivica Crnkovic and Per Runeson. Adopting a component-based software architecture for an industrial control system—a case study. *Component-Based Software Development, Lecture Notes in Computer Science*, 3378:232–248, 2005.

Lars Mathiassen, Andreas Munk-Madsen, Peter Axel Nielsen and Jan Stage. *Objektorienterad analys och design*, page 223. Studentlitteratur, 2001.

R. Jon McGee and Richard L. Warms. *Anthropological Theory*, chapter Ethnoscience and Cognitive Anthropology, pages 385–386. The McGraw-Hill Companies, third edition, 2004.

Douglas McIlroy. Mass produced software components. In Peter Naur and Brian Randell, editors, *Software Engineering: Report on a conference sponsored by the NATO Science Committee, Garmisch, Germany, 7th to 11th October 1968*, pages 138–155. North Atlantic Treaty Organization, NATO Scientific Affairs Division, 1969. Available at `http://homepages.cs.ncl.ac.uk/brian.randell/NATO/nato1968.PDF`. The PDF version's pagination differs from the original; refer to pages 79–87 in this version.

Merriam-Webster. Merriam-Webster's Online Dictionary: Definition for *object*, 2009a. Available at `http://www.m-w.com/dictionary/object`.

Merriam-Webster. Merriam-Webster's Online Dictionary: Definition for *component*, 2009b. Available at `http://www.m-w.com/dictionary/component`.

Microsoft. *Behind the Code—Life and Times of Anders Hejlsberg.* Video interview, 2005. Available at `http://channel9.msdn.com/shows/Behind+The+Code/Life-and-Times-of-Anders-Hejlsberg/`.

Microsoft. *Behind the Code—Tony Williams: Co-inventor of COM.* Video interview, 2006. Available at `http://channel9.msdn.com/shows/Behind+The+Code/Tony-Williams-Co-inventor-of-COM/`.

Kevin Mukhar, Chris Zelenak, James L. Weaver and Jim Crume. *Beginning Java EE 5: From Novice to Professional.* Apress, 2005.

Peter Naur and Brian Randell, editors. *Software Engineering: Report on a conference sponsored by the NATO Science Committee, Garmisch, Germany, 7th to 11th October 1968*, 1969. North Atlantic Treaty Organization, NATO Scientific Affairs Division. Available at `http://homepages.cs.ncl.ac.uk/brian.randell/NATO/nato1968.PDF`.

Greg Olsen. From COM to common. *ACM Queue*, 4(5):20–26, June 2006. Available at `http://queue.acm.org/detail.cfm?id=1142043`.

Open Software Foundation. *OSF DCE Version 1.1: DCE Porting and Testing Guide*, November 1995. Available at `http://www.opengroup.org/dce/download/`.

George Orwell. *Nineteen Eighty-Four.* Penguin Group, centennial edition, 2003. Original version published in 1949.

David L. Parnas. On the criteria to be used in decomposing systems into modules. *Communications of the ACM*, 15(12):1053–1058, December 1972.

Ted Pattison. Basic instincts: Porting applications from MTS to COM+. *MSDN Magazine*, March 2000. Available at `http://msdn.microsoft.com/en-us/magazine/cc301351.aspx`.

A. J. Perlis. A new policy for algorithms? *Communications of the ACM*, 9(4):255–256, April 1966.

Erik Persson. *Shadows of Cavernous Shades: Charting the Chiaroscuro of Realistic Computing.* Ph.D. thesis, Department of Computer Science, Lund University, 2002.

Cuno Pfister and Clemens Szyperski. Why objects are not enough. In *Proceedings, First International Component Users Conference.* SIGS Publishers, 1996.

Jeff Prosise. Windows 2000: Asynchronous method calls eliminate the wait for COM clients and servers. *MSDN Magazine*, April 2000a. Available at `http://msdn.microsoft.com/en-us/magazine/cc301334.aspx`.

Jeff Prosise. Implementing handler marshaling under Windows 2000: DeviceClient sample app. *MSDN Magazine*, August 2000b. Available at `http://msdn.microsoft.com/en-us/magazine/cc302323.aspx`.

Jeff Prosise. Understanding COM apartments, part I. *CodeGuru*, 2001. Available at http://www.codeguru.com/cpp/com-tech/activex/apts/article.php/c5529.

Ingo Rammer and Mario Szpuszta. *Advanced .NET Remoting*. Apress, 2005.

Manos Renieris and Steven P. Reiss. Almost: Exploring program traces. In *Proceedings of the 1999 Workshop on New Paradigms in Information Visualization and Manipulation*, pages 70–77. ACM Press, 1999.

Dennis M. Ritchie. The evolution of the Unix time-sharing system. *Lecture Notes in Computer Science*, 79:25–35, 1980. Available at http://www.cs.bell-labs.com/who/dmr/hist.html.

Miro Samek. Portable inheritance and polymorphism in C. *Embedded Systems Programming*, 10(12), December 1997. Available at http://www.embedded.com/97/fe29712.htm.

Douglas C. Schmidt and Steve Vinoski. Object interconnections: Real-time CORBA, part 1: Motivation and overview. *C/C++ Users Journal*, December 2001. Available at http://www.ddj.com/cpp/184403809.

Douglas C. Schmidt and Steve Vinoski. CORBA metaprogramming mechanisms, part 1. *C/C++ Users Journal*, March 2003. Available at http://www.ddj.com/cpp/184403860.

Douglas C. Schmidt and Steve Vinoski. The CORBA Component Model: Part 1, evolving towards component middleware. *C/C++ Users Journal*, February 2004. Available at http://www.ddj.com/cpp/184403884.

Douglas C. Schmidt, Nanbor Wang and Steve Vinoski. Object interconnections: Collocation optimizations for CORBA. *C++ Report*, September 1999.

Stephen B. Seidman. *IFIP International Federation for Information Processing*, volume 280, chapter The Role of Professional Societies in the Emergence of Software Engineering Professionalism in the United States and Canada, pages 59–67. Springer Boston, 2008.

Alan Snyder. Encapsulation and inheritance in object-oriented programming languages. In *Proceedings of the OOPSLA '86 Conference*, pages 38–45. Association for Computing Machinery SIGPLAN, 1986.

Ian Sommerville. *Software Engineering*, chapter Component-Based Software Engineering. Pearson Education, eighth edition, 2007.

Mark Stoodley, Kenneth Ma and Marius Lut. Real-time Java, part 2: Comparing compilation techniques. IBM developerWorks, April 2007. Available at http://www.ibm.com/developerworks/java/library/j-rtj2/.

Bjarne Stroustrup. *The Design and Evolution of C++*. Addison-Wesley, 1994.

Bjarne Stroustrup. Multiple inheritance for C++. *The C/C++ Users Journal*, May 1999.

Kevin J. Sullivan. Designing models of modularity and integration. In George T. Heineman and William T. Councill, editors, *Component-Based Software Engineering: Putting the Pieces Together*, pages 341–367. Addison-Wesley, 2001.

Sun Microsystems. *OpenOffice.org 2.3 Developer's Guide*, 2007. Available at http://api.openoffice.org/docs/DevelopersGuide/DevelopersGuide.pdf.

Sun Microsystems. *Being Unique with Sony Ericsson.* Presentation from the JavaOne conference, 2009. Available at http://java.sun.com/javaone/2009/general_sessions.jsp.

Symbian Foundation. *Introduction to the ECom Plug-in Architecture*, 2008. Available at http://developer.symbian.org/main/documentation/carbide/ (search for "ecom").

Clemens Szyperski, Dominik Gruntz and Stephan Murer. *Component Software: Beyond Object-Oriented Programming.* Addison-Wesley, second edition, 2002.

Darryl K. Taft. Distributed OSGi effort progresses. *Ziff Davis eWEEK*, 2008. Available at http://www.eweek.com/c/a/Application-Development/Distributed-OSGi-Effort-Progresses/.

Koji Taniguchi, Takashi Ishio, Toshihiro Kamiya, Shinji Kusumoto and Katsuro Inoue. Extracting sequence diagram from execution trace of Java program. In *Proceedings of the 2005 Eighth International Workshop on Principles of Software Evolution (IWPSE'05).* Institute of Electrical and Electronics Engineers, 2005.

Dave Templin. Simplify app deployment with ClickOnce and registration-free COM. *MSDN Magazine*, April 2005. Available at http://msdn.microsoft.com/en-us/magazine/cc188708.aspx.

Will Tracz. COTS myths and other lessons learned in component-based software development. In George T. Heineman and William T. Councill, editors, *Component-Based Software Engineering: Putting the Pieces Together*, pages 99–113. Addison-Wesley, 2001.

Andrew Troelsen. *Pro C# with .NET 3.0.* Apress, 2007.

Doug Turner and Ian Oeschger. *Creating XPCOM Components.* Brownhen Publishing, 2003. Available at https://developer.mozilla.org/en/Creating_XPCOM_Components.

Jon Udell. ComponentWare. *BYTE Magazine*, pages 46–56, May 1994.

Bill Venners and Bruce Eckel. Delegates, components, and simplexity: A conversation with Anders Hejlsberg, part III, 2003. Available at http://www.artima.com/intv/simplexity3.html.

Mark Vigder. The evolution, maintenance, and management of component-based systems. In George T. Heineman and William T. Councill, editors, *Component-Based Software Engineering: Putting the Pieces Together*, pages 527–553. Addison-Wesley, 2001.

Steve Vinoski. CORBA: Integrating diverse applications within distributed heterogeneous environments. *IEEE Communications Magazine*, 35(2), February 1997. Available at http://steve.vinoski.net/ieee.pdf.

Padmal Vitharana. Risks and challenges of component-based software development. *Communications of the ACM*, 46(8):67–72, August 2003.

Rainer Weinreich and Johannes Sametinger. Component models and component services: Concepts and principles. In George T. Heineman and William T. Councill, editors, *Component-Based Software Engineering: Putting the Pieces Together*, pages 33–49. Addison-Wesley, 2001.

James E. White. A high-level framework for network-based resource sharing. In *AFIPS '76: Proceedings of the June 7-10, 1976, national computer conference and exposition*, pages 561–570, 1976. Also available as RFC 707, at `http://tools.ietf.org/html/rfc707`.

The veracity of all Internet addresses has been verified on January 30, 2010.